Master Degree Project

# Investigation of Methods for Testing Aspect Oriented Software

Master Degree Project in Informatics
Advanced level, 30 ECTS
Spring term 2013

Kallol Banik

Supervisor: Birgitta Lindström
Examiner: Jonas Mellin

# Abstract

Aspect-oriented programming is a comparatively new programming paradigm which intends to overcome some limitations that approaches such as procedural programming and object-oriented programming have. Traditional approaches are unable to properly capture some design decisions. Aspect-oriented programming introduces some new properties that we don't find in the structural programming or object-oriented programming. New design patterns of aspect-oriented software introduce new fault types and new challenges for testing. Testing is an important part in the software development to produce quality software. Research on testing aspect-oriented software has been going on for several years but it still remains to invent testing approaches that cover all features of aspect-oriented software. This dissertation surveys test methods for aspect-oriented software and presents a comparison among the testing methods which reveals the strengths and weaknesses of current methods for testing of aspect-oriented software. This comparative overview of proposed test methods can be helpful for testers who intend to test aspect-oriented software. The conclusion presents the research contribution of this dissertation and proposes future work.

**Key words:** Aspect-oriented programming, software testing, fault model**.**

# Contents

**1 Introduction** **7**

1.1 Dissertation Outline …….. 7

**2 Background** **8**

2.1 Aspect-oriented programming concept ……..... 8

2.2 Software testing ……... 11

2.3 Aspect oriented software testing ……... 14

**3 Problem definition** **17**

3.1 Motivation …… 17

3.2 Aim and objectives ……17

3.3 Research method: Literature survey and analysis..……18

**4 Survey Test methods** **20**

4.1 Test methods based on data and flow graph...……...20

4.1.1 Data flow based unit testing…….. 20

4.1.2 Aspect flow graph for testing …… 21

4.1.3 Inter-procedural Aspect Control Flow Graph based testing……..21

4.2 Test methods based on state models..……22

4.2.1 State-Based Approach……...22

4.2.2 State-based incremental testing of aspect-oriented programs ……..23

4.2.3 Testing aspect-oriented programs with finite state machines…… 23

4.3 Test methods based on UML model …….. 24

4.3.1 A Model-Based Approach to Test Generation…… 24

4.3.2 Testing aspect-oriented programs with UML design models……24

4.3.3 Testing AOS using UML activity diagram…… 25

4.4 Test methods based on mutation …….. 25

4.4.1 Mutation testing of pointcuts …… 25

4.4.2 Automated generation of pointcut mutants…… 26

4.4.3 Mutation testing for aspect-oriented programs…… 26

4.4.4 AjMutator: A Tool For the mutation analysis……. 27

4.5 Testing approaches based on formal methods ……...27

4.5.1 Aspect-oriented programming with model checking…… 27

4.5.2 Towards a practical approach to Test Aspect-Oriented Software……28

4.6 Testing methods based on JUnit Framework…… ……..28

4.6.1 Unit-testing aspectual behavior…… 28

4.6.2 Testing as object-oriented programs…… 29

4.7 Random testing …… 29

4.7.1 On the applicability of Random Testing…… 29

**5 Analysis of the testing methods 31**

5.1 Fault types that the proposed testing methods target……..…….. 31

5.2 Approach……..…….. 34

5.3 Goal of the proposed testing methods …….. 38

**6 Syntheses of the testing methods……41**

6.1 Pair wise analyses between Fault type and Approach……42

6.2 Pair wise analysis between Fault type and Goal……..47

6.3 Pair wise analysis between Approach and Goal…… 56

**7 Discussions 61**

**8 Related Work 63**

**8 Conclusion 66**

8.1 Research Contributions …… 66

8.2 Future Work …… 66

# List of Figures

1. Cross-cutting concerns (Sommerville, 2010) ..... 9

2. Construction of an Aspect with AspectJ. (Sommerville, 2010) ..... 10

3. Aspect Weaving (Sommerville, 2010) ..... 11

4. Example of presence of fault in a small function (from course lecture) ..... 12

5. RIP-model ( Voas, 1995) ..... 13

# List of Tables

1. Table 1: Testing Methods for aspect-oriented software target fault..... 33

2. Table 2: The relation between the aspect-oriented software testing methods and traditional techniques .....36

3. Table 3: Goal of the proposed testing methods .....39

4. Table 4(A)**:** Pair wise analysis between fault type and approach..... 42

5. Table 4(B)**:** Pair wise analysis between fault type and approach in details..... 43

6. Table 5(A): Pair wise analysis between fault type and goal.....47

7. Table 5(B): Pair wise analysis between fault type and goal in details.....48

8. Table 6(A): Pair wise analysis between approach and goal..... 56

9. Table 6(A): Pair wise analysis between approach and goal in details..... 57

10. Tabel7: Comparison between the related papers and current dissertation..... 48

# Introduction

In our everyday life we are depending on computer systems such as cars and telephones for different activities. The architecture of a system might be complex in such a way that a change in a small part of the system may affect other parts of the system. Sometimes it is difficult to identify the specific part of the software where a modification in requirements applies. Aspect-oriented software is an approach in software development that addresses some of these problems.

Aspect-oriented programming is one of the recent programming paradigms. Aspect-oriented programming introduces new properties that we do not find in the structural programming or object-oriented programming. Therefore, there are some new types of faults in aspect-oriented programming which are not present in any other programming paradigm. These new fault-types bring new challenges to testing of aspect-oriented software. Testing is an essential part of the software development process to produce high quality software. We know that half or more software development cost is related to testing activities (Collofello & Vehathiri, 2005). This dissertation presents an investigation over testing methods of aspect-oriented software to provide a better idea applicable testing method and the issues related in testing aspect-oriented programming.

## 1.1 Dissertation Outline

Section 2 provides necessary background information for the dissertation which contains aspect-oriented programming concept, software testing and aspect-oriented software testing. Section 3 defines the problem and presents aim, objectives and also method for how to achieve the aim of the dissertation. A motivation for the proposed aim of the dissertation is given, which also contains that why the dissertation is necessary and who needs it. Section 4 presents the literature survey over testing methods for aspect-oriented software. Analysis among the testing methods for aspect-oriented software is presented in section 5 where a comparison between different testing methods is presented according to different categories. Section 6 presents related work. Finally, section 7 presents conclusions including research contributions and future work.

# 2 Background

This section presents some fundamental concepts which are necessary background for the dissertation. Section 2.1 introduces some important aspect-oriented programming concepts. From section 2.2, we come to know about definition and model of software testing. Finally, section 2.3 gives a brief concept about testing aspect-oriented software.

## 2.1 Aspect-oriented programming concept

In 1997, aspect-oriented programming (AOP) was started at Xerox's PARC laboratories with the development of the AspectJ programming language (Sommerville, 2010, page 584). Aspect-oriented programming is a relatively new programming paradigm that offers advantage such as separation of cross-cutting concerns. Aspect-oriented programming allows for better modularity, cohesion, understandability, maintainability of a program or system (Parizi & Ghani, 2007).

According to Jacobson and Ng (2004), concerns reflect the system requirements and priorities of stakeholders in the system. For example, train braking can be a *functional concern* in a train control system. Performance, reliability and availability reflect *quality of service concerns*. Security and safety are very important *policy concerns*. Maintainability and configurability are *system concerns* because they are related to attributes of the system.

Key principle of software design and implementation is the separation of concerns. It means that one should organize the software so that each element in the program, for example class, method, procedure, etc. does one thing and one thing only (Sommerville, 2010, page 267). It becomes easier to anyone to understand each part of the program through concerns, without having knowledge about other elements. It also makes easy to focus any part of the program. If any change of requirement is needed, then it can become easier to trace the specific part of the program.

Procedural programming approach and object-oriented programming approach is unable to properly capture some limitations for design decisions that the program must implement. These two approaches are facing problems while dealing with a special type of concern that cut across other concerns. Aspect-oriented software engineering indicates an approach to software development that overcomes this limitation. When a change is implemented to the requirements in a large system, it may affect several other components. Aspect-oriented programming addresses this problem by separation of cross-cutting concerns. The goal is a system that is easier to reuse and maintain.

Figure 1 shows cross-cutting concerns. If we consider a hospital patient information system, we find some core concerns like new patient requirement, medication requirements and patient management requirements. These core concerns are associated with the system's primary purpose. This system also requires security requirements according to the hospital information security policy and recovery requirement to ensure that the data is not lost in case of system

failure. Here, security requirement and recovery requirement are cross-cutting all other system core requirements as they influence the implementation of core requirements like new patient requirement and medication requirement.

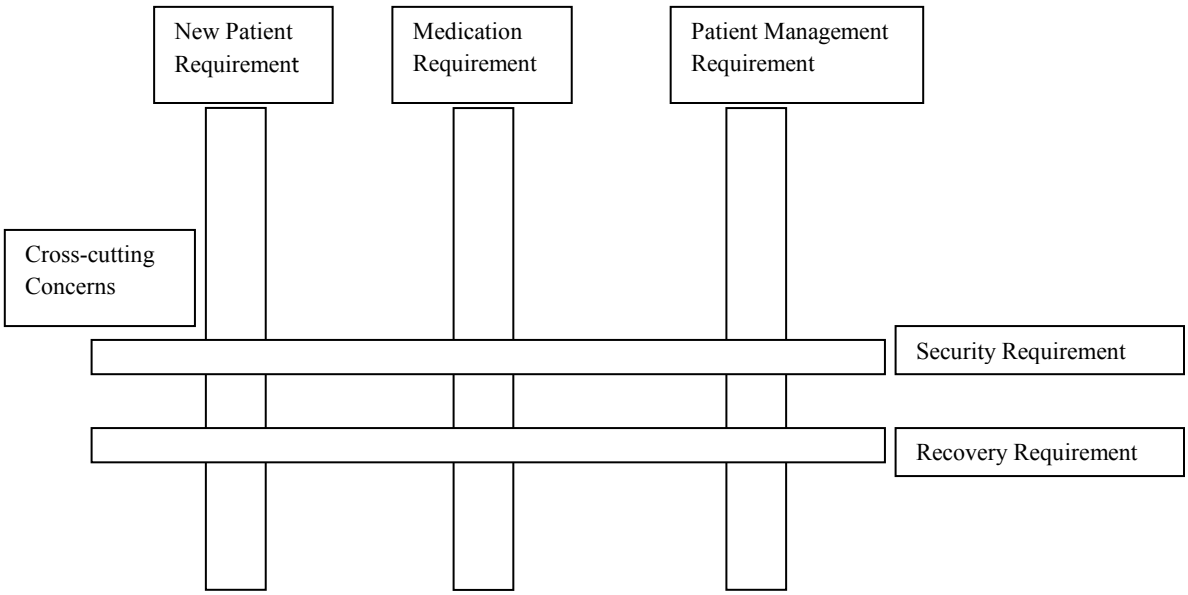


**Figure 1** Cross-cutting concerns (Sommerville, 2010)

Aspect-oriented programming introduces some new concepts and abstractions. New type of concepts and abstractions of aspect-oriented software are stated below, which we do not find in any other programming paradigm.

An *aspect* is known as an abstraction that encapsulates all handling related to a concern (Zhou et al., 2004). An aspect implements system functionality and includes a specification of where the cross-cutting concern is to be woven into the program, and code to implement that concern. There is a clear difference between aspect and other program abstractions in that the aspect itself includes a specification of where it should be executed (Sommerville, 2010, page 573). According to Parizi & Ghani (2007), an aspect is a construct that encapsulates a cross-cutting concern which is similar to class in object oriented programming.

Figure 2 shows the construction of a confirmation aspect. This aspect confirms extra security for information database. It may happen that the responsible person has left his or her computer logged on. At that time if any other person wants to change data from that computer, the system requires to check that the operator is the right person to make any change of the database. It is a security policy of the system. In case of updating any information the person need to re-authenticate him/her to the system. If the password does not match, the system will log out.

*Pointcut* is a part of aspect which is a statement that defines where the aspect will be woven into the program (Sommerville, 2010, page 574). Pointcut is also known as a set of patterns that is used to select join point. (Parizi & Ghani, 2007). In figure 2, first line of aspect confirmation indicates a pointcut. This means that the code after the pointcut should execute to any method whose name starts with update. The character wildcard ('*') indicates that there may be any sequence of characters.

```
aspect confirmation
{
   before: call (public void update* (..)) //  pointcut
   {
      // advice which executes when woven into the executing system
      int value = 0 ;
      string userPassword = Password.Get ( value ) ;
      while (value < 3 && userPassword != thisUser.password ( ) )
      {
          // 3 values to get the right password
          value = value + 1 ;
         userPassword = Password.Get ( value ) ;
      }
      if (userPassword != thisUser.password ( )) then
        System.Logout (thisUser.uid);
     }
}
```

**Figure 2**: Construction of an Aspect with AspectJ. (Sommerville, 2010)

After the pointcut, there is code which is known as advice. The *advice* implements the cross-cutting concern. The *advice* is a method like construct that contains additional behavior to be added at the matched joint point. The advice is woven into the join points when a pattern of 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. (Alexander et al., 2004). Three kinds of advice objects are well known: before advice, after advice, and around advice. In case of *before advice*, its body executes before the body of the matched join point. *After advice* executes after executing the body of matched join point. According to Alexander et al. (2004), 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.

*Join point* is used by pointcuts to specify where code should be executed and it is a distinguishing characteristic of aspects. To realize what pointcut means, we need to understand another concept—the idea of a join point. A join point reflects an event which occurs during the execution of a program; so, it could be a method call, the initialization of a variable, the updating of a field, etc (Sommerville, 2010, page 574). Join point is a well-defined point in the execution

of a program. Examples of join points are calling or execution of methods, access to an attribute, and initialization of an object (Parizi & Ghani, 2007).

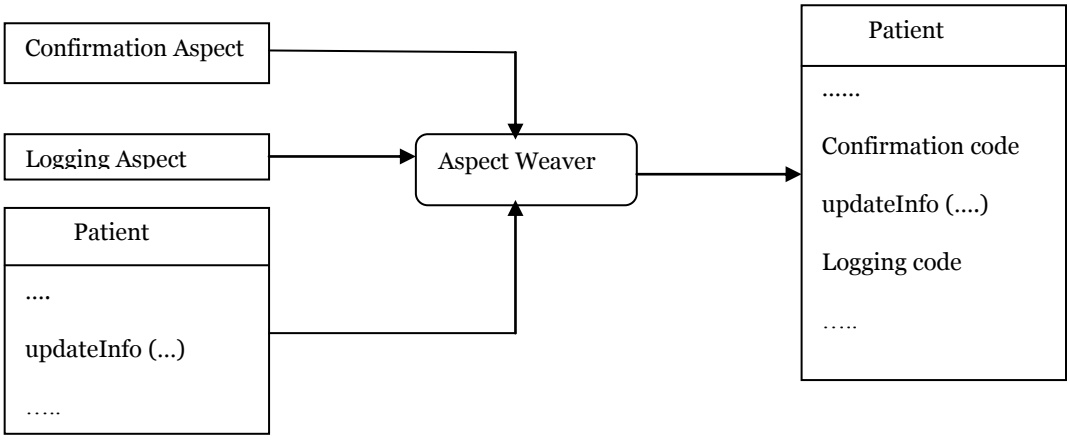


**Figure 3:** Aspect Weaving (Sommerville, 2010)

Behavior on aspects is merged to the original code by the *aspect weaving* process. We can say that weaving is a process of merging the cross-cutting concerns to the core concepts of a system or program. Aspect weaving composes the code of the base classes in AspectJ and the aspects ensure that applicable advice runs at the appropriate join point. Figure 3 shows an illustration of aspect weaving. The responsibility of an aspect weaver is to merge the advice at the join points specified in the pointcuts. Aspect weaver in the figure is an extension to compiler that processes the definition of aspects and the object classes and methods that define the system. Then a new program is generated by the weaver with the aspects included at the specified join points. The correctness of the weaving process is proved by executing cross-cutting concerns at the right places in the final system (Sommerville, 2010, page 575).

## 2.2 Software testing

Software testing is essential to ensure software quality. Testing activities consume at least 30 percent of the project effort in most software projects. Software testing can consume between 50 to 80 percent of project effort in safety critical applications (Collofello & Vehathiri, 2005). In spite of being costly and time-consuming process, software testing is essential if a high quality product is to be produced (Collofello & Vehathiri, 2005).

There are lots of researchers have defined software testing in different ways. According to Lindström (2000) testing is the act of executing the system in order to find faults and increase the confidence of the system. It can, however, never prove that the system is correct. As we know

100% testing is not possible. Researchers are still experimenting to find out testing techniques that cover most of the possible faults for specific system and that is the way to increase confidence of the system. Therefore the above definition seems that it gives a better idea about software testing.

Software testing should reveal faults. Many faults are design mistakes that propagate. Inserting wrong input by an operator is also known as fault. A fault is also a static implementation mistake made by the programmer. If the execution reaches a fault, it may enter an incorrect internal state that is an error. Such error may propagate to an externally incorrect behavior. That is called a failure. Test cases should provide clear feedback so that revealed faults can be easily corrected (Ammann & Offutt 2008, Beizer 1990).

```
 public int findMax (int[] y)
// Effects: If y==null throw NullPointerException
// else return the biggest element in y.
{
      int x=y[y.length-1];
      for (int i=y.length-1; i > 0; i--)
      {
            if (y[i] > x)
            {
                  x=y[i];
            }
      }
      return x;
}
```

**Figure 4:** Example of presence of fault in a small function (from course lecture)

In figure 4, we can see a small function which finds maximum value from an input array. In the "for" statement there is a fault that is i > 0. For this condition it cannot check the first element. But the array should be checked from the last element to first element. The fault resides in the condition i>0. The correct expression should have been i==0. There would be an exception if input is null. The execution does not reach the fault if the input is null. So fault is not triggered by the input. Therefore, the execution does not reach the error state and no failure occurs. If we give the input array like [2,4,6,8]. The result would be 8. In this case execution reaches the fault but it will not propagate to failure. The function will give correct output because the maximum value is not in the first position. When we will give an input array like [8,6,4,2], the result would be 6. The function did not check the first element. We observe here an incorrect behavior. In this case execution reaches the fault, also the error stage is reached and failure is observed.

*Testability* is a quality aspect of the software and very hard to assess. Although important for the testers and the quality of the system, the testers seldom have this knowledge. Testability reveals the probability that software will fail in the next execution during testing if the software contains a fault. Testability also reflects the scenario of how likely the testing will reveal the fault. Therefore, when a software engineer finds a software artifact with low testability then he/she

already comes to know that the testing alone is unlikely to result in a satisfactory of verification and alternative verification is required. Test engineer also can try to improve the testability of the software artifact, which is a matter of design (Ammann & Offutt, 2008, page 284). Hence, tester should be involved in the design. Testability problem was casted in terms of controllability and observability which comes originally from hardware testing. Friedman (1991) and Binder (1994) adapted the condition testability in software testing.

In order to distinguish between correct and erroneous behavior of the software, it is necessary to observe why, when and what the software does. Observability indicates observe capability for both internal and external behavior during test execution. Behavior of software can be estimated when it is observed properly. However, the software must facilitate such an observation, that is, it must be observable (Schütz 1994). Observability reveals about the incidents of the software when it executes. Observability indicates the ability of observe what is happening in the software while execution.

Software controllability shows how easy it is to provide a program with the needed inputs, in terms of values, operators and behaviors (Ammann & Offutt, 2008, page 14). Another property which is known as reproducibility is related with the property controllability. Controllability is a property. Controllability is the extent to which the system allows a tester to control the execution of tests. Controllability reveals how much influence over software a tester has while executing test cases. Reproducible indicates that the software's behavior will remain same while executing the same test cases. Otherwise we cannot become sure about the error which is responsible for the failure. So reproducibility is controllability's part.

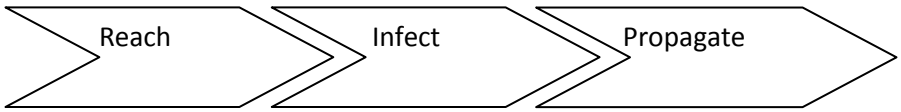


**Figure 5:** RIP-model (Voas, 1995)

Figure 5 shows necessary three conditions for observing a failure. There are different approaches for tackling testability. Already we are familiar with a model of testability built on the fault, error and failure. As the key components of testability, some authors have focused on software observabiliy and software controllability. RIP-model was introduced by Jeffrey Voas in 1995 where he defined testability in terms of three attributes: reach ability, infection and propagation. Figure 5 shows the RIP-model. If we see the small function example in figure 4, we find that reach indicates that the execution of the function should reach the fault. Fault reflects the incorrect statement. So the execution becomes infected which also propagate incorrect output.

The sufficiency of tests for the program under test is also an important consideration while testing. To increase confidence while testing, test cases should test enough portions of the

program. The coverage criteria define the sufficiency. Coverage criteria are used to decide which test inputs to use because we cannot test with all inputs. Therefore, effective use of coverage criteria makes testing also effective and provides informal assurance that the software is of high quality and reliable. Coverage criteria also show when to stop testing.

An essential part of any viable software development process is regression testing. Regression testing means re-testing software that has been modified, which constitutes vast majority of testing efforts in commercial software development. Most of the time, any small change in software may influence a large part of the system. Regression testing is used to verify that the change does not lead to any new problem (i.e., all tests that passed before the change should pass after it.) Software testing reflects the presence of faults only, not the absence of faults (Dahl et al. 1972).

## 2.3 Aspect-oriented software testing

According to Sommerville (2010, page 586), the problems of inspecting and deriving tests for aspect-oriented programs are a significant barrier to the adoption of aspect-oriented software development in large software projects. For distinct characteristics and design development design, aspect oriented software introduce new challenges for testing. We find some new types of faults in this new technology for software development.

When aspects are woven into the classes and methods of core concern, there are some complex behaviors of both core and cross-cutting concerns. Alexander et al. (2004) indentify some characteristics of aspects. Independent identity or existence is absent for aspects. For identity and execution context, aspect rely on the other classes. Any changes of those classes may influence the aspects. When aspects are woven into any application it also creates some changes in control flow and data flow structures. Complexity in the behavior of the aspect increases the possibility of presence of some new kind of faults. Alexander et al. (2004) also identify some source of faults for the woven aspects in the program. A fault may be present in the core concern. So an error may occur without the weaving process. There may be a fault in the aspect code but isolated from the woven context. When multiple aspects are woven into the primary abstraction, it becomes difficult to find the location of the fault.

For some special design criteria (e.g., weaving process) of aspect oriented software development process, testing also requires some specialties. Comparatively this kind of development process is immature. We find some approaches for effective testing techniques for aspect oriented software. Zhao (2003) has proposed data-flow based unit testing of aspect-oriented programs. Xu et al. (2004) propose a state based approach and an aspect flow graph for testing aspect-oriented software. Bernardi and Lucca (2007) proposed an inter-procedural aspect control flow graph for testing aspect-oriented software. Alexander and Bieman (2004) proposed a fault model for systematic testing of aspect-oriented software.

Object-oriented programming has some characteristics such as static state and encapsulation. These characteristics determine that object-oriented programming cannot change the internal

behavior of an object and it brings down the flexibility ratio of software development (Li & Xie, 2009). The testing of Aspect Oriented Programs should focus on the faults likely to appear due to the nature of this software development paradigm (Ceccato et al., 2005). Aspect is enough capable to affect the behavior of a certain class. The type of faults that can occur in object-oriented program also can occur in the core concerns in aspect-oriented programming. In aspect-oriented program, we find some new kinds of faults for aspect-oriented features. Some fault types are discussed below:

**Problem with pointcut's strength:** Correctness in the expression of pointcut is very important for the proper weaving process. Pointcut is a part of aspect, which contains specifications to match a particular type of join point to be intercepted. So it is important for the developers to write the pointcut with proper strength. Otherwise two kinds of scenarios may occur: 1) If the pattern of the pointcut is too strong then it would be unable to select some required join point. 2) If the pattern of the pointcut is too weak then it will select some additional join point which should be ignored (Moreira et al., 2010). Both the above scenarios cause aspect to fail by intercepting wrong join point. Therefore, to test the correctness of the strength of the pointcut is very important. Test correctness of the pointcut indicates the validation of the pointcut designators that define the execution point for the aspects.

**Conflicts between aspects:** We can consider a system which is developed by aspect-oriented programming. When an aspect is added it means that new behavior is added into the system. The new aspect may break previous tests. If the newly added aspect conflicts with an old aspect then it affects the correctness of the old aspect. Hence, old aspects need to be re-tested after the addition of a new aspect. This occurrence affects the systems quality (Moreira et al., 2010). So it is important to find the source of the conflict which is caused by adding new aspect. Sometimes programmers make mistakes carelessly while defining pointcuts. As a result a piece of advice may influence during another advice execution. Therefore an aspect becomes the cause for changing behavior of another aspect. If both the incorrectly matched advices become active while execution, it may cause an infinite loop. It is very difficult to find out this kind of bug because it looks correct (Zhang & Zhao, 2007). So when a new aspect is added into the system it affects some code block. But the matter is to find out the code block which is affected. Then it will become easy to specify the portion of the code which should be retested.

**Absence of advice precedence:** In aspect-oriented programming we find that more than one advice can intercept in a single join point. The result of the program may depend on the sequence of execution of the advices. The missing of the execution precedence may affect the program result without any error report. This occurrence may be a failure. Therefore, order of advice execution should be maintained while more than one advice intercept in a common joint point (Zhang & Zhao, 2007). When a new aspect is introduced, it may also alter the flow of execution. For the new aspect, new fields can be added in the class of a base system.

**Bugs created while weaving process:** Sometimes we don't find any presence of a bug in both aspect and base class. But while weaving process bug may appear. It is also possible that the bug was hidden into the aspect or the base class (Ubayashi & Tamai, 2002). Because of the complexity in the behavior of aspect-oriented programming, it becomes very problematic to find the location of the bug. Sometimes this kind of bug is created for wrong weaving order. Still

there is insufficient knowledge for the testers to cover most common errors, for the complexity behavior of the aspect-oriented programming.

According to Alexander et al. (2004), the unique characteristics of AOPs do not occur in object-oriented, or even procedure-oriented, programs. Each characteristic has the possibility to manifest new fault types that ultimately lead to program failures.

As stated above, separation of concern is a key principal of aspect-oriented programming. The possibility of testing separate concerns separately can make aspect-oriented programming testing easier than traditional application (Ceccato et al., 2005). The authors also address two problems to test the aspects separately: 1) the way to test the base code isolate. 2) The way to re-test the code when a new aspect is introduced.

One of the development goals of any kind of software is to ensure high quality. Proper testing is important to ensure the quality. Testing activities often consumes a large portion of the cost for software development. While developing or inventing techniques, it is therefore required to find out cost effective testing techniques. Therefore, there are new challenges to test aspect oriented software.

# 3 Problem definition

The aim of the dissertation is to provide an analytical overview over current testing methods for aspect-oriented software. A motivation for the dissertation is presented in section 3.1 and a set of objectives that need to be fulfilled in order to reach the aim for this thesis is given in section 3.2. Research method is presented in section 3.3.

## 3.1 Motivation

Overview over current testing methods is useful in the sense that it points out strengths and weaknesses with different approaches by comparison. This information may help testers decide on how to test aspect-oriented software. Furthermore, an overview can point out new research areas as well as limitations of current research.

Research in testing of aspect-oriented software is still going on. Some researchers (e.g., Zhou et al., 2004) are proposing only a first step towards a comprehensive approach for testing aspect-oriented software. We also get some testing approaches for aspect-oriented program (e.g., Anbalagan & Xie, 2006) which only highlights some specific features. Aspect-oriented software introduced some new kinds of faults which we did not see in procedural program and object-oriented program. There is no coherent overview of the strengths and weaknesses of different test methods which can help testers and researchers to get idea about the current methods. There are some surveys, which discuss the effectiveness of some of the test methods. The effectiveness of the test methods is examined by their capability of fault finding. But until now, more test methods for aspects-oriented software are proposed. Therefore, it is a requirement for the testers and researchers also to know that whether the currents methods are able to cover all the specific fault types for aspect-oriented software or not. We have not found any research, which reveals not only the fault finding capabilities of the proposed testing methods of aspect-oriented software but also other strengths and weaknesses.

## 3.2 Objectives

To achieve the aim of the project there are some objectives, which need to be fulfilled. The objectives are as follows:

**Objective 1: Survey test methods**

By survey in the test methods, we can collect information about the current test methods of aspect-oriented software. Structural presentation of that information can provide overview over state-of-art test methods. Achieving the first objective will provide a summary of the state-of-arts. When the survey will show a summary over state-of-art test methods, then it will provide a

good idea about the aspect-oriented software testing. To achieve the aim of the project, the first objective will fulfill the primary requirements.

**Objective 2: Identify a set of criteria to categorize the testing approaches.**

A categorization of the testing approaches is required to get a clear overview among them. Achieving the second objective will give an overview of state-of-arts and forms a basis for an analysis. To provide an overview, we need a categorization that is useful for (i) presentation and (ii) as a basis for an analysis. The categorization criteria should focus on aspects of the methods that are interesting from a tester's perspective, for example fault types.

**Objective 3: Analyze the testing approaches for aspect-oriented software.**

Testing approaches for aspect-oriented software will be analyzed in such a way that reflects contrasting and combining results from different studies of the approaches. This can reveal some interesting findings, relationships or disagreements among the results. The analysis of the approaches will fulfill or complete the overview over testing methods of aspect-oriented software. Analytic representation of the testing methods of aspect-oriented software helps to achieve the aim of the project and provide an overview that is useful in the sense that a tester can make an informed decision on which combination of test methods to use for a system.

## 3.3 Research method: Literature survey and analysis

A literature survey or literature review indicates that one should read and report on what the literature in the field has to say about a specific topic or subject. We can also say that it is a study and review of relevant literature materials in relation to a topic that one have been given. To reach the aim of the proposed dissertation I have chosen literature survey as methodological approach. One of the benefits of conducting a literature survey is that an overview over state-of-art test methods is given. An alternative survey method is to take an interview among the software companies. But as the aspect-oriented technology is comparatively new, it is not practiced widely in the industrial sectors recently. Therefore, literature survey can be proper research method to get the valid result for the dissertation while there are a good number of papers about testing methods of aspect-oriented software.

Conducting a survey also requires proper searching methodology to cover all the testing methods for aspect-oriented software. Otherwise, we cannot trust it as an up-to-date survey. Literature survey always follows a way of identifying, evaluating and interpreting all available relevant research on a particular topic. To survey testing methods for aspect-oriented software, a proper searching method can ensure that everything is covered. For the current survey three kind of searching strategy is followed which are stated below:

1. There are a good number of databases where we can search for the research articles and papers on computer science. For the current survey, chosen databases are: 1) IEEE

Xplore, 2) ACM Digital Library, and 3) Springer Link. The reason behind choosing these databases is that each of the databases contains a comprehensive archive of the organization's journals, magazines, and conference proceedings. Each of the databases is well-known for containing prestigious, academic and scholarly articles, especially, in the field of computer science. Search terms are: 1) “Aspect-oriented software testing”, 2) “Aspect-oriented software verification”, 3) “Testing approaches for aspect-oriented software”, 4) “Testing methods for aspect-oriented software”. To select the papers from the proposed databases, at first, it is examined that the title is ok. Then, the abstract of the paper is read. If the abstract also support the current research, then the paper is read.

2. Each paper which presents a testing method for aspect-oriented software also contains the references at the end of the paper which is also a reasonable source to find out related papers. like the previous example, when the abstract of a specific paper present a proper methodology for testing aspect-oriented software then it is included into the survey.

3. There are some related works for each of the papers that present testing methods for aspect-oriented software. While reading the abstract of those related papers, when it seems that any paper present a methodology to test aspect-oriented software then the paper is included into the survey.

# 4 Survey Test methods

In this section, result of the survey for testing methods for aspect-oriented software is presented. The testing approaches are classified below, in order to make a clear sense about the testing approaches. Following classification gives an overview over test methods of aspect-oriented software. Every method is divided into two paragraphs. First paragraph reflects the related traditional techniques and goal of the method, where second paragraph reflects targeted fault types of the proposed methods.

## 4.1 Test methods based on data and flow graph

In the following papers, test cases are evaluated from the graphical representation of the system. Both control flow and dataflow graphs are used for the representation of the system. The graph abstraction is a well-defined and well-studied structure of the programs (Ammann & Offutt 2008). Control flow graphs show how the control is transferred in a program. In testing, these used to identify input parameters that execute all paths through the graph according to some criteria (e.g., at least all edges should have been covered at least once). An extension to these, data flow graphs, argument control flow graphs with data flow information. The data flow graphs are typically covered with respect to paths or combination of definitions (e.g., assignments of variables) & uses (of these variables).

### 4.1.1 Data flow based unit testing

Zhao (2003) provides data flow based testing approach for aspect oriented software. In this paper, the proposed testing approach tests two types of units for aspect-oriented program. The units are the aspects that are modular units of cross-cutting implementation of the program and the classes whose behavior may be affected by one or more aspects. Control flow graph (CFG) is constructed to obtain data flow information in a module of an aspect with related classes or a class with related aspect. The control flow relationships in the module are presented by the control flow graph. An inter-procedural control flow graph (ICFG) is constructed to obtain data flow information that involves more than one module in aspect or class. The ICFG is composed from a group of CFGs each of which represents a module in the aspect or class. Framed control flow graph (FCFG) is constructed to obtain data flow information from an aspect or a class. There are a collection of CFGs each of which presents a module in the aspect or class and some additional arcs in the FCFG. Three types of control flow graphs are presented to build the data flow based unit testing.

According to Naqvi et al.(2005), data-flow-based unit testing strategy does not target any particular fault type, but it seems reasonable to reveal the faults like-failure to preserve state invariants because state based faults are usually associated with erroneous data flow. The fault type is proposed in the fault model of Alexander et al. (2004). Behavior control or data dependences of a method can be changed by around advises. Therefore, change should be visible

in def-use pairs (Parizi & Ghani, 2007). The authors demands that the proposed testing method handle unit testing problems that are unique but does not present any empirical study in a large project.

### 4.1.2 Aspect flow graph for testing

Aspect flow graph based testing (Xu et al., 2004) deals with combining state models (class and aspect) and flow graphs (methods and advices) as an aspect scope coverage model of the system for producing test suites. The Proposed method presents a hybrid testing model that combines a responsibility-based testing model and an implementation-based testing model. The testing approach which is presented in the paper consists combination of aspect sate model and class state model into an aspect scope state model (ASSM). Aspect Flow Graph (AFG) is constructed by merging ASSM and the flow graphs for methods and advices. Next, the authors construct a transition tree which represents a collection of high level behaviors. At last, a set of code based test cases are generated for each of high level class behaviors. Kumer et al. (2010) stated "*Constructing a flow graph for an Aspect oriented program will make it possible to apply a number of white box testing strategies and test coverage criteria includes Statement coverage, Branch coverage and Data flow testing coverage".* They also mentioned that test cases for regular classes can be reused and the testing effort can be further reduced while the class code is tested already by focusing only on the new behavior and the aspects

The flow graph based testing approaches for AOPs can detect two types of faults in the candidate fault model of Alexander et al. (2004) which are: 1) incorrect focus of control flow and 2) incorrect changes in control dependencies. But Naqvi et al. (2005) have proposed that this testing approach is not effective in detecting faults of incorrect focus of control flow because Aspect Flow Graph is a model of static weaving and the specific fault arise due to the dynamic behavior of aspect-oriented program. And any empirical study of the proposed method in large scale project is not presented.

### 4.1.3 Inter-procedural Aspect Control Flow Graph based testing

Bernardi & Lucca (2007) propose a set of coverage criteria that is based on the interactions among the advices and methods they affect. Proposed method deals with the failures which occur in the interactions among advices and class methods due to faults in pointcut expressions and advices. As the proposed testing approach focuses on the interactions among aspects and classes, therefore we can classify the approach as integration testing. The proposed approach aims to cover all the relationships which are created due to the implementation of static and dynamic cross-cutting features. The inter-procedural aspect control flow graph (IACFG) is created to satisfy the proposed criteria. IACFG represents the inter-procedural interaction among aspects' advices and classes' methods. IACFG depicts how the code of an advice is woven into the code of a method. It also shows the points in the control flow of an operation where the execution control is transferred to another operation. There are a set of control flow graphs which are known as Operational Control Flow Graph (OCFG). IACFG is constructed by a set of OCFG. The coverage criteria include all the structural elements of an IACFG. The coverage

criteria have been derived from the traditional criteria like node coverage, branch coverage, condition coverage and multiple conditions coverage criteria which are used for white-box unit testing.

According to Kumer et al.(2009), the proposed model can reveal following fault from the fault model of Alexander et al. (2004) and Cecato et al. (2005): incorrect strength in pointcut patterns, failure to preserve static invariants, incorrect focus of control flow, incorrect changes in control dependencies and incorrect changes in exceptional control. The effectiveness of the proposed method is supported by an empirical study in large scale project.

## 4.2 Test methods based on State models

In the state model, a system is represented by its different states and the triggering conditions for transitions between these states. The following testing methods are based on state models of aspect-oriented software. State based testing analyze that the state machine is complete and consistent enough for model and implementation testing.

### 4.2.1 State-Based Approach

The state based approach suggested by Xu et al., (2004) is directly built upon a test design pattern for object-oriented program. An aspectual state model (ASM) of the system is developed for a testable model of class behaviors along with additional advices defined in aspects, which are attached dynamically when specific join points are reached. ASM is extended from FREE (Flattened Regular Expression) state model for object-oriented software development. FREE has similarities with the state model in UML but definitions are not found in UML. To test aspect-oriented program according to the state-based approach, FREE is extended into ASM for specifying both classes of the core concern and aspects of the crosscutting concerns. Then ASM is transformed into a transition tree, which implies a test suite for adequate testing of the object behavior and interaction between classes and aspects in terms of message sequence. The testing method is extended into transition tree based testing to deal with crosscutting concerns. In the transition tree, every path from the root to a terminal leaf node is a test requirement for testing object behaviors. If the variables are assigned specific values that satisfy the corresponding conditions, then the test requirement becomes as test case. For the adequate test of the aspect, every related path of the tree should be exercised at least once, which test the intended behavior of the aspects. Multi-conditional coverage is applied in the transition tree. A test suite achieving N+ coverage can reveal all state control faults and many state bugs. For the reason of improper use of advices, pointcuts and joint points, the interaction between aspects and classes may fail. There are two advantages for ASM: 1) it presents better model to present aspect-oriented software and 2) following the model to test aspect implementation is more effective.

The proposed testing method can reveal some faults like incorrect strength of pointcut pattern and failure to preserve state invariants. These two types of faults are proposed in the fault model

of Alexander et al. (2004). According the authors, transition tree based method provides a sound support for aspect-oriented software development without negative effects on the development process. An empirical study in large scale project is presented to support the effectiveness of the proposed method.

### 4.2.2 State-based incremental testing of aspect-oriented programs

Xu & Xu (2006) propose a test method for aspect-oriented software where the base class tests are reused for testing aspect-oriented programs. For this purpose, aspects are viewed as incremental modifications to their base classes. Incremental approach to test aspect-oriented software can reduce testing cost because the test cases are reused and it also beneficial to localize programming problems by identifying aspect-specific faults. In the proposed system testing approach, at first, the state models of the base classes are built. Then, abstract test cases are generated from the base model. Afterwards, it is suggested to instantiate and abstract test cases to form concrete test suites for the base classes. Then, the base classes are tested. Consequently, aspect models are built and they are weaved into the base models. Model of the system can be represented by control flow graph. Afterwards, test suits are generated for the aspect-oriented program as a whole by modifying, reusing and extending base class test cases and instantiating new abstract test cases. Finally, the aspect-oriented program is tested.

The fault types which are detected by the proposed testing approach are: 1) pointcut expression picking out extra join points, 2) pointcut expressions missing join point and 3) incorrect advice types. In the result of an empirical study of the proposed method, it is found that majority of the base class tests can be reused for aspect testing. Though, some of them require modification. Authors claim that the incremental testing approach is similar to traditional regression testing but the difference is that the aspects as a structured way to specify modifications make it feasible to investigate systematic reuse and modification of the existing tests. Any empirical study of the proposed method in large scale project is not presented.

### 4.2.3 Testing aspect-oriented programs with finite state machines

Xu et al. (2010) present a framework which is called MACT (Mode-based Aspect/class Checking and Testing). MACT is used for testing whether or not aspect-oriented programs conform to their aspect-oriented state models. In the proposed testing strategy, we find combination of structure-oriented testing strategies and property-oriented testing strategies for generating aspect tests from aspect oriented state models. Structure-oriented strategy use coverage criteria to derive test, on the other hand, property-oriented strategy generate tests from the counterexamples of model checking. To derive the testing process, the specification of a property is used by the property-oriented testing. The goal of the property-oriented testing is to validate a program with respect to a large property .While checking an aspect-oriented state model; mutation analysis is used to evaluate for the fault detection capabilities. In this paper, classes and aspects of an aspect-oriented program is modeled with finite state machine which is similar to the existing work on testing object-oriented program with state machines. Finite state machine can be represented by control flow graph. State model of an aspect-oriented program

consists of aspects models, class models and a precedence relation of the aspects. The aspect-oriented state model captures the behaviors of the base classes and the aspects on the state transitions of their base classes. While implementing both structure-oriented strategy and property-oriented strategy to test aspect-oriented program, the authors demand that integrating different strategies can be a more effective approach with state models.

From the empirical study of the proposed testing strategy in large scale project it is found that the different kinds of faults are detected. The detecting fault types are: incorrect pointcut strength, incorrect aspect precedence, incorrect advice type and faults due to intertype declarations.

## 4.3 Test methods based on UML model

UML model based testing methods for aspect-oriented software are presented in this section. UML includes a set of graphic notation techniques to create visual model of aspect-oriented software. In case of UML based testing, test requirements are derived from UML analysis artifacts such as use cases, their corresponding sequence and collaboration diagram.

### 4.3.1 A Model-Based Approach to Test Generation

Model based approach to test generation for aspect-oriented programs (Xu & Xu, 2005) presents interaction between aspects and classes. The proposed model is based on aspect-oriented UML model. The model of aspect-oriented program consists aspect diagram, class diagram and sequence diagrams. Crosscutting concerns are expressed by aspectual class diagram where each aspect consists of two components: pointcut and advice. In the aspectual class diagram, port represents join point and connector represents pintcut expression. Methods' classes and aspects' advices are specified by sequence diagrams. A flow graph is constructed in a bottom-up way. Then, the flow graph is expanded into a flow tree. The flow graph is constructed in such a way that the test generation starts from the end of the sequence diagram. The successors of each node are based on polymorphic coverage and branch coverage. Each path from a leaf to the root of the flow tree represents one run of the woven sequence diagram, which indicates a test case. Improper use of join points, pointcuts and advices for the separate concern can cause failure in the interactions between classes and aspects. The proposed approach is a preliminary report of the authors' ongoing research on model based testing of aspect-oriented programs.

The proposed testing approach does not target any fault type of the fault model of Alexander et al. (2004) and Cecato et al. (2005). There is no empirical study is presented against the testing approach.

### 4.3.2 Testing aspect-oriented programs with UML design models

Xu et al. (2008) present a UML-based approach to test aspect-oriented programs that test whether or not an aspect-oriented program conforms to its expected crosscutting behavior. UML design model of the system derives tests for exercising interactions between aspect and classes. There are class diagram, aspect diagram and sequence diagram in each aspect-oriented model, which describe the static structure and dynamic behavior respectively. An aspect-object flow (AOF) tree is generated from the sequence diagram and class/aspect diagrams. At first AOF graph is constructed from the sequence diagram, then the AOF graph is converted into AOF tree. Each path from the root to a leaf of an AOF tree reflects an abstract message sequence which also indicates a template of test cases. Condition coverage, polymorphic coverage and loop coverage is applied into the AOF tree. The conditional coverage indicates that both the true and false conditions of a conditional behavior are exercised. From the polymorphic coverage we see that a polymorphic message affected by aspects is exercised with each subclass of the class in the polymorphic message.

The authors also present an empirical study in large scale project that shows that model based testing approach is capable of revealing several fault types. The fault types that the proposed approach targets: incorrect pointcut strengths, incorrect aspect precedence and incorrect advice type.

### 4.3.3 Testing AOS using UML activity diagram

Kaur & Grag (2012) used UML activity diagram to test aspect-oriented software. Activity diagram represents flow graph, which shows flow from one activity to another. Activity diagram of aspect-oriented software captures important features like join points, pointcut, advice etc. There are both base model and aspect model in the aspect-oriented model. The proposed testing approach consists of three steps. In the first step, activity diagram of the base model is built and the test sequence of base model is generated. Therefore, the faults which are related with the base model are eliminated and complexity is reduced. In the second step, aspects are integrated with the base model and the test sequence is generated. Execution of sequence is verified in the third step by comparing the actual sequence with the expected sequence.

Fault types like incorrect advice type, strong or weak pointcut and incorrect aspect precedence can be revealed by this testing approach. In the paper, authors present an empirical study where they implement the model into a ATM system and find faults.

## 4.4 Test methods based on mutation

In this section, mutation based testing methods for aspect-oriented software are presented. Mutation testing is created to verify the correctness of the code. Testers insert faults into the program under test to verify the program. Mutation testing is known as fault-based testing technique which is used to infect faults into an existing program.

### 4.4.1 Mutation testing of pointcuts

Mutation testing of pointcuts (Anbalgan & Xie, 2006) is based on a framework that automatically generates mutants for a pointcut expression and identifies mutants that seems close of the original expression. Pointcut match a join point. There are two steps in mutation testing of pointcuts: 1) to create effective mutants for a pointcut expression and 2) to test the mutants using the designed test data. Pointcuts use signature that include various designators, wildcards and their combination with logic operators to select a join point. The pattern in the signature should have enough strength to select the proper join point. When the pattern is too strong then the poincut is unable to select all necessary join points. When the pattern of the poincut is too weak then it selects additional join point that should be ignored.

The testing process which is presented here is applicable only to verify the strength of pointcut. Proposed approach is implemented in large scale project. Preliminary Results reveal strong, weak and neutral mutant.

### 4.4.2 Automated generation of pointcut mutants

Anbalagan & Xie (2008) have developed a new framework that automatically identifies the strength of each pointcut expression and generates pointcut mutants with different strength. The proposed approach provides an overview of the framework. The framework is developed to identify automatically the strength of each pointcut and generates poitcut mutants The framework consists six components: 1) Pointcut Parser that identifies pointcuts in the given AspectJ source code, 2) Join Point Candidate Identifier that identifies the join point candidates from the given Java bytecode for the base program, 3) Mutant Generator that forms mutants for the pointcuts, 4) Pointcut Tester that verifies the join point candidates, 5) Mutant Classifier that identify the type of pointcut mutant in comparison with the original pointcut, 6) Distance Measurer that helps developers identifying pointcut mutants, which resemble closely the original pointcut. The generated pointcut mutants are also classified by the framework. The framework ranks the mutants based on similarity measure. Inspecting the ranked list of piontcut mutants along with their strength differences, developers can choose the mutants for conducting mutation testing.

This testing approach is also applicable only to verify the strength of pointcut. An empirical study in large scale project is conducted to support the effectively of the framework on pointcuts.

### 4.4.3 Mutation testing for aspect-oriented programs

Ferrari et al. (2008) propose mutation based testing approach for aspect-oriented programs where they design a set of mutation operators for AspectJ-based programs. First of all, a set of aspect-oriented fault types are identified, which are based on previous works on aspect-oriented program testing. Then, they design a set of mutation operators which is based on the identified fault types and which is also applicable in different similar languages. They present a

relationship between aspect-oriented implementations and fault types for AspectJ, Spring AOP, JBoss AOP and CasearJ language. An analysis is also presented for identified fault types for generalization to aspect-oriented implementations other than AspectJ; and there are descriptions about the application of the proposed operators. The proposed mutation operators cover a wide range of the fault types.

The paper presents four kind of aspect-oriented fault types: 1) pointcut related faults, 2) ITD (inter-type declarations) related faults 3) advice related faults; and 4) base program related faults. Proposed mutation operators are divided into three groups: 1) operators for pointcut expressions, 2) operator for general declarations; and 3) operators for advice definitions and implementations. The proposed operators cover many aspect-oriented fault types except the faults related to incorrect changes in class structures. The future work of the project is to assessing the effectiveness of the operators in simulating faults.

### 4.4.4 AjMutator: A Tool For the mutation analysis

Delamare et al. (2009) introduce a tool which is named AjMutator. AjMutator is used for the mutation analysis of AspectJ pointcut descriptors. Introducing faults in the pointcut generators, AjMutator can generate mutants. It also executes a set of test cases on the generated mutants. AjMutator also capable for classification of the mutant pointcut descriptors by comparing the sets of join points matched by the mutant and the initial pointcut descriptors. While manual selection of the mutants is quite difficult, automatic classification offers great benefits.

This testing approach is also applicable only to verify the strength of pointcut. AjMutator is very effective on large scaled system because the automatic classification is helpful when large numbers of mutants are generated. The automatic classification of the equivalent mutants helps to avoid the unnecessary execution of mutants and allows a more accurate mutation score.

## 4.5 Testing approaches based on formal methods

Formal methods for testing aspect-oriented software are presented below. Formal methods are known as particular kind of techniques for specification, development and verification of the system.

### 4.5.1 Aspect-oriented programming with model checking

Ubayashi & Tamai (2002) have proposed an automatic verification approach using model checking. Model checking verifies the woven program that contains any unexpected behavior such as deadlocks. To verify the correctness of concurrent finite automata, model checking has been used in the proposed method. The main objective of the method is to use model checking to verify the correctness of aspect-oriented programs. Other objective is to provide a framework for aspect-oriented program based model checking. Model checking is a formal method, which is used to check whether a structure of a system satisfies a property or not. In the proposed

method, Kripke structure is used as a structure for model checking. Kripke structure is a type of state transition graph. In the proposed method, model checking can be used while runtime testing. But unfortunately, the checking framework, which is provided, cannot cover all testing features.

The proposed testing approach does not target any fault type of the fault model of Alexander et al. (2004) and Cecato et al. (2005). There is no empirical study is presented to support the approach.

### 4.5.2 Towards a practical approach to Test Aspect-Oriented Software

Zhaou et al. (2004) have introduced a formal approach to test aspect-oriented software. According the authors the proposed method proposes a first step towards a practical approach to test aspect-oriented software. An algorithm is also developed to identify test cases that are relevant to aspects under test. A new testing coverage definition is also proposed to specify the sufficiency of test cases to test aspects. Finally a tool is developed to automate test case selection and coverage calculations. To test aspect-oriented software, the proposed approach adapts traditional unit testing, integration testing and system testing. The authors suggest that the test cases, which are developed for regular classes can be reused. Reusing relevant test cases will also reduce the cost of testing and thereby test cases also achieve higher efficiency. The proposed algorithm is based on control flow analysis that is used to select relevant test cases to test aspects. When reused test cases are unable to cover the aspects, then, new test cases should be developed. The tool is developed so that it can help to select relevant test cases and calculate the test coverage for the aspects. The tool is developed with the Eclipse environment which is an open source Java integrated development environment.

An empirical study is presented to support the method. The proposed testing approach does not target any fault type of the fault model of Alexander et al. (2004) and Cecato et al. (2005).

## 4.6 Testing methods based on JUnit Framework

JUnit is known as a unit testing framework for the Java programming language. JUnit is very popular for test-driven development. In this section, proposed testing methods are based on or extended from JUnit framework. JUnit framework is used by developers to implement unit testing in Java and accelerate programming speed and increase the quality of code

### 4.6.1 Unit-testing aspectual behavior

Lopes & Ngo (2005) present a testing approach that focus on testing aspectual behavior which is implemented in pieces of advice. The proposed testing approach is based on the Java Aspect Markup Language (JAML). Unit testing of aspectual behavior classes is conducted using JamlUnit which is an extension of JUnit. We already know that JUnit is a widely used framework for the unit testing of the Java program. A set of Java helper classes is consisted by the JamlUnit that enable programmers to write regular JUnit test cases for testing aspect code as independent units. The proposed testing technique focuses on the faults that reside in aspectual implementation and independent from the execution context. But the limitation of the approach is that testing aspects in isolation cannot detect faults resulting from the integration of aspects with the implementation of core concerns. Therefore, the authors claim that unit testing should be followed by integration testing. On the other hand, the contribution of the current approach is that it allows the possibility and requirements for devising clean unit testing techniques for aspect oriented programs.

Proposed testing approach does not target any fault type of the fault model of Alexander et al. (2004) and Cecato et al. (2005). There is no empirical study is presented against the testing approach.

### 4.6.2Testing as object-oriented programs

Zhao & Alexander (2007) publish a paper where they discuss the feasibility of testing the woven code and propose an object-oriented program (OOP) testing method for AspectJ programs. Proposed method intends to test the decompiled woven code. The authors think that the proposed method is a feasible and effective method. The advantage of the proposed method is that there is no need to take account of the new language features brought by aspect-oriented programming. The decompiled code is Java source code that is tested using normal OOP testing techniques. JUnit is a simple framework which is used to test decompiled code. CodePro AnalytiX is an automated Java testing tool which is used to generate the JUnit test framework for the decompiled java code.

Proposed testing approach does not target any fault type of the fault model of Alexander et al. (2004) and Cecato et al. (2005). The empirical study of the proposed method is based on simple programs. Therefore, more complicated experiments are needed to justify the proposed method.

## 4.7 Random testing

The only example for random testing of aspect-oriented software is presented in this section. According to Hamlet (1994) random testing has capability of replacing human choice with chance selection. Random testing is quite simple and is not particularly costly in comparing with systematic approaches.

### 4.7.1 On the applicability of Random Testing

Parizi et al. (2009) provide a framework for random testing of aspect-oriented programs. Proposed testing approach has potentiality in terms of test case generation and implying reliability and statistical estimates. There are five major stages in the proposed framework for random testing which outlines the abstract architecture of the approach. In the first stage, aspects with those methods whose behavior may be affected by the aspect's advice are identified in the target system. Then, the expected behavior of aspect is specified. In the second stage, random input is generated and selected for aspects under test, which can be started with the base code or directly to the aspects and related classes. The authors have developed an infrastructure to generate inputs by a joint point model. Expected behavior is specified in the third stage, which are going to be checked while runtime execution and should be examined what conditions the aspect must meet at certain points of the execution. In the fourth stage, an infrastructure is developed in which the test process can be continued or recovered in spite of any failure occurring during the test running. The outcome is assessed to determine pass or fail of a given test run in the fifth stage. The authors claim that proposed approach is the first attempt in the community that applies random testing on aspect-oriented programming.

Proposed testing approach does not target any fault type of the fault model of Alexander et al. (2004) and Cecato et al. (2005). There is no empirical study is presented against the testing approach.

# 5 Analysis of the testing methods

In this section, an analysis of the testing methods for aspect-oriented software is presented. The analysis reflects contrasting and combining results from different studies of the approaches which may reveal some interesting findings, relationships or disagreements among the results. The analysis is presented on the base of some important criteria about the testing methods which are mentioned before in chapter 4 and specified below.

1) Fault types that the proposed testing methods target.
2) Approach.
3) Goal of the proposed testing methods.

## 5.1 Fault types that the proposed testing methods target

In this section, an investigation is conducted to find out the fault types that the methods target. To do that, it is required to specify the faults for aspect oriented software at first. This investigation will also provide an overview about the effectiveness of the approaches. We shall have to consider that all the proposed approaches do not target the specific fault types for the aspect-oriented software. Weyuker (1993) explores a concept that effectiveness of a method can be defined as the number of faults the method will find. Weyuker (1993) also says that the effectiveness of a test method is only possible to measure if one can compare two methods for the same software. Weyuker (1993) says that uniform relations that guarantee one criterion is better at fault detection than another, according to certain types of well-understood probabilistic measure of effectiveness, are especially valuable. Alexander and Bieman (2004) defines a candidate fault model based on reasoning about how AOP's work. Cecato et al. (2005) extend the model of fault types. In the fault model, Alexander et al. (2004) specify six types of faults. Cecato et al. (2005) add three types. Some of the papers, which are published recently identify a new fault type that is known as incorrect advice type. Total ten types of faults for the aspect-oriented software are stated below where first six are specified by Alexander et al (2004).

1. **Incorrect strength in pointcut patterns:** Pointcuts contain specification that select particular type of join points according to a signature that includes a pattern. If the pattern of the specific pointcut is too strong, then it is unable to select some necessary join points. On the contrary, if the pattern of the pointcut is too weak, then it selects some additional join points that should be ignored. Any one of the above occurrence may cause incorrect behavior of the woven target. This kind of fault causes the aspect to fail. So a test of aspect is required in this case.

2. **Incorrect aspect precedence:** It may happen that multiple advices from multiple aspects are woven into a common join point. In this case, it is important to specify the precedence or control the order of the advice while advices are going to be woven into the common join point. There is possibility of unpredictable behavior if no precedence is assigned to the aspects. To reveal this kind of fault, testing of weave orders is required.

3. **Failure to establish expected post-conditions:** When an advice is woven into a core concern, it may cause for any change that shows deviation from the post-conditions. But the clients of the core concerns expect that the core concerns should behave according to their contacts. Therefore it is challenge for the aspect developers to establish expected post-conditions. Core concerns method may fail for this kind of fault. Re-test of the core concerns which are connected in weaving process is required here.

4. **Failure to preserve state invariants:** The physical representation and the methods that act on the state of a concern define the behavior of a concern. It is always mandatory to ensure that the state invariants are satisfied to keep the behavior of the concern constant. Weaving process may cause violation to preserve state invariants. So it is another challenge for aspect developers and another source of errors. This type of fault also causes core concerns methods to fail.

5. **Incorrect focus of control flow:** A pointcut designator selects a method's join points to be woven. At weave time, this selection is determined. Sometimes it requires proper information to make such a decision at run time. Because sometimes joint points require to be selected in a particular execution condition. Therefore, if the execution fails to maintain the proper context then it causes a failure which is difficult to diagnose. Because of being active of any advice in wrong time cause this kind of fault. Condition coverage is suggested to reveal this kind of fault.

6. **Incorrect changes in control dependencies:** Around advice may change the execution of the matched join point body, or may even replace the matched join point body. Thus the original control flow structure can be changed. New branches alter the dependencies among statements and new data also may be inserted. This fault will also affect core concern behavior like fault no 3 and 4.

7. **Incorrect changes in exceptional control flow**: Advices cause an implicit modification in the system control flow. Specially, for those advices which are containing statements that possibly throw an exception. An exception triggers the execution of an appropriate catch statement. Different branches might be taken in the original code, when such a construct is used to modify the system exception handling.

8. **Failure due to inter-type declarations**: Because of inter type declaration, static cross cutting modifies the base code. This mechanism may produce ripple effects in the control flow. For example, the Boolean operation x instance of y produces a true value and the dynamic type of the object x is a subtype of y. If an aspect changes the inheritance relation of x then the previous Boolean operator may return a different value which is also altering the control flow.

**Table 1**: Testing Methods for aspect-oriented software target fault type ( "×"= targeted fault type by the method, "-"= not yet determined).

| Testing Methods \ Fault Type | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Data flow based unit testing (Zhao, 2003) | | | | × | | | | | | |
| Aspect flow graph for testing (Xu et al., 2004) | | | | | | × | | | | |
| Inter-procedural Aspect Control Flow Graph based testing (Bernardi et al., 2007) | × | | | × | × | × | × | | | |
| State-Based Approach (Xu et al., 2004) | × | | | × | | | | | | |
| State-based incremental testing of aspect-oriented programs (Xu & Xu, 2006) | × | | | | | | | | | × |
| Testing aspect-oriented programs with finite state machines (Xu et al., 2010) | × | × | | | | | | × | | × |
| A Model-Based Approach to Test Generation (Xu and Xu, 2005) | - | - | - | - | - | - | - | - | - | - |
| Testing aspect-oriented programs with uml design models. (Xu et al., 2008) | × | × | | | | | | | | × |
| Testing AOS using uml activity diagram (Kaur & Grag, 2012) | × | × | | | | | | | | × |
| Mutation testing of pointcuts (Anbalgan & Xie, 2006) | × | | | | | | | | | |
| Automated generation of pointcut mutants (Anbalagan & Xie, 2008) | × | | | | | | | | | |
| Mutation testing for aspect-oriented programs (Ferrari et al., 2008) | × | | | | | | | × | | × |
| AjMutator: A Tool For the mutation analysis (Delamare et al., 2009) | × | | | | | | | | | |
| Aspect-oriented programming with model checking (Ubayashi & Tamai 2002) | - | - | - | - | - | - | - | - | - | - |
| Towards a Practical Approach to Test Aspect-Oriented Software (Zhaou et al., 2004) | - | - | - | - | - | - | - | - | - | - |
| Unit-testing aspectual behavior (Lopes & Ngo, 2005) | - | - | - | - | - | - | - | - | - | - |
| Testing as object-oriented programs (Zhao & Alexander, 2007) | - | - | - | - | - | - | - | - | - | - |
| On the applicability of Random Testing (Parizi et al., 2009) | - | - | - | - | - | - | - | - | - | - |

9. **Incorrect change in polymorphic calls:** In the period of weaving process, when a method introduction is used to override a method inherited from a super class, at that time it may occur modification in the system behavior. In the case of after aspect weaving, the specific method becomes related with method introduction which was redirected to the method in the super class.

10. **Incorrect advice type:** It may happen that the advice which is specified in design period it is changed in the implementation time. Such an example, in design period, an around advice was specified but in the implementation period after advice executes. Unconsciousness of the developer may cause this kind of fault.

In section 4, targeted fault types of the proposed methods are mentioned in the second paragraph of each description of the testing methods. Targeted fault type for each testing method is identified in table 1 according to that information. Table 1 shows that none of the testing approaches target the following fault types: 3 (Failure to establish expected post-conditions) and 9 (Incorrect changes in polymorphic calls). Ten test approaches target the first fault type-incorrect strength in pointcut patterns. Second fault type (Incorrect aspect precedence) and fourth fault type (Failure to preserve state invariants) are targeted by three testing approaches. Fifth fault type (Incorrect focus of control flow) and seventh fault type (Incorrect changes in exceptional control flow) are target by a common testing approach (Bernardi et al., 2007). Sixth fault type (Incorrect changes in control dependencies) and eighth fault type (Failure due to inter-type declarations) are targeted by two different testing approaches. Tenth fault type (Incorrect advice type) is targeted by five testing approaches. Among the testing methods, Inter-procedural Aspect Control Flow Graph based testing method (Bernardi et al., 2007) targets highest number (5) of fault types. Some of the proposed testing methods do not target any one of the fault types.

## 5.2 Approach

In this section, an investigation is conducted to find out that is there any relation between the traditional techniques for sequential software and the proposed testing approaches of aspect-oriented software. From the survey of the testing methods, already we know that some of the testing approaches of the aspect-oriented software are based on traditional techniques and some traditional techniques can be derived from the models of aspect-oriented software which are presented to test.

Sequential software can be represented by four types of model according to Amman & Offutt (2008). Four types of model are: graph based, logic based, input domain model (IDM) and syntactic structures. Considering the models of the software, test methods also can be categorized according which model they are based on. The test methods are stated below briefly.

1. **Graph based coverage:** Most widely used testing coverage criteria are based on graphs. Graph based coverage shows the way to evaluate test set according the paths

corresponding to the test cases "cover" a system's graphical representation. Graph based coverage criteria can be divided into two types: 1) control flow coverage criteria, 2) data flow coverage criteria. In 1975, Legard invented control flow graphs (Legard & Marcotty, 1975). Control flow coverage criteria are also known as structural graph coverage criteria. Node coverage and edge coverage are known as fundamental criteria for structural coverage criteria. Node coverage which is also known as statement coverage, execute each statement of sequential software at least once. In many commercial testing tools, node coverage is used. On the other hand the test requirement of edge coverage is that it would execute the each branch of sequential software at least once. At first, the reference of data flow testing was found in a technical report of Osterweil and Fosdick in 1974 (Osterweil & Fosdick, 1974). Data flow criteria shows that the values are carried from definition (where value of a variable is stored) to use (variable's value is accessed). Harrold and Soffa have applied data flow testing to integration testing (Mary Jean Harrold & Mary Lou Soffa, 1991). In object-oriented software, Alexander, Offutt, Buy, Orso and Pezze applied data flow testing to inheritance and polymorphism ( Alexander & Offutt 1999, Buy et al. 2000 ). Graph coverage also can be derived from use case and Finite State Machines (FSMs). Data flow coverage criteria are not applicable to derive graph from use case. While node coverage and edge coverage are mostly used to derive graph from use case (Amman & Offutt, 2008, chapter 2).

2. **Logic based coverage:** In spite of being quite old technique, the use of logic based coverage has gradually increased in recent time. US Federal Aviation Administration (FAA) has incorporated logic expressions for safety critical avionics software in commercial aircraft. Logic expressions can be derived from decisions in program, FSMs, state charts and requirements. Predicate is known as an expression that evaluates to a Boolean value and clause indicate a predicate without logical operator. There is a lack of subsumption between predicate coverage and clause coverage. Decision point of the program reflects the predicates. To apply logic expression becomes problematic when there are large numbers of clauses in the predicate. Wise programmers tend to write predicate with not more than three clauses. When a predicate contain only single clause, all logic coverage criteria collapse to predicate coverage. One of the disadvantages of the logic coverage is that sometimes it becomes hard to apply logic criteria to program source. In case of applying logic coverage criteria to FSMs, several issues should be understood and dealt to create predicate and test values (Amman & Offutt, 2008, chapter 3).

3. **Input Space Partitioning:** Different researchers have identified input space partitioning as domain testing, category partition, boundary value analysis, partition testing and include equivalence partitioning. Balcer, Hasling and Ostrand first described category partition method in 1988 (Balcer et al., 1989). Input space partitioning shows the systematic ways to divide the input space according the test requirement. Input domain reflects the possible values of the input parameters. This type of testing process has several advantages. No automation is required for testing. Testers also require small amount of training. But the testers should have clear knowledge about input domain and the description of inputs. Input space partitioning can be equally applied in unit,

**Table 2**: The relation between the aspect-oriented software testing methods and traditional techniques ("×"= related traditional technique, "-"= not yet determined)

| Traditional Methods / Testing Methods | Graph based coverage | | Logic based coverage | Input Space Partitioning | Syntax-Based Testing |
|---|---|---|---|---|---|
| | CFC | DFC | | | |
| Data flow based unit testing (Zhao, 2003) | | × | | | |
| Aspect flow graph for testing (Xu et al., 2004) | × | × | | | |
| Inter-procedural Aspect Control Flow Graph based testing (Bernardi et al., 2007) | × | | × | | |
| State-Based Approach (Xu et al., 2004) | × | | × | | |
| State-based incremental testing of aspect-oriented programs (Xu & Xu, 2006) | × | | | | |
| Testing aspect-oriented programs with finite state machines (Xu et al., 2010) | × | | | | × |
| A Model-Based Approach to Test Generation (Xu and Xu, 2005) | × | | | | |
| Testing aspect-oriented programs with uml design models. (Xu et al., 2008) | × | | × | | |
| Testing AOS using uml activity diagram (Kaur & Grag, 2012) | × | | | | |
| Mutation testing of pointcuts (Anbalgan & Xie, 2006) | | | | | × |
| Automated generation of pointcut mutants (Anbalagan & Xie, 2008) | | | | | × |
| Mutation testing for aspect-oriented programs (Ferrari et al., 2008) | | | | | × |
| AjMutator: A Tool For the mutation analysis (Delamare et al., 2009) | | | | | × |
| Aspect-oriented programming with model checking (Ubayashi & Tamai 2002) | × | | | | |
| Towards a Practical Approach to Test Aspect-Oriented Software (Zhaou et al., 2004) | × | | | | |
| Unit-testing aspectual behavior (Lopes & Ngo, 2005) | - | - | - | - | - |
| Testing as object-oriented programs (Zhao & Alexander, 2007) | - | - | - | - | - |
| On the applicability of Random Testing (Parizi et al., 2009) | - | - | - | - | - |

integration and system testing. It is also easy to adjust procedure to do more or fewer test. To consider separately of the each parameter of the domain is a common way to apply input space partitioning. Then it is required to partition the possible values into blocks. Afterwards, variables for each parameter should be combined. This process is known as input domain modeling. According to the creativity and experience, different testers can apply this model in different way. But they shall have to be conscious about increasing the quality of the input domain model. A partition should maintain two conditions: 1) it should contain the total domain which is known as completeness and 2) the blocks should not be overlapped which is known as disjoint. We find two approaches for input domain modeling. One is interface-based approach and another is functional based approach. The interface-based approach shows how to directly develop characteristics of the input parameters of a program which is under test. The function-based approach shows how to develop characteristics of the functions or behavioral view of the program which is under test. Testers should be careful about implementing input domain modeling because fault can be revealed by selecting invalid type of argument to a function (Amman & Offutt, 2008, chapter 4).

4. **Syntax-Based Testing:** In syntax-based testing, tests are created from the syntax of the software artifact to identify the correctness. The syntax can be expressed according to some sort of grammar. Syntactic description can come from programs, integration elements, design documents and input descriptions. In spite of being expensive, mutation is a very effective test criterion. Mutation is very effective for programs. Mutation operators are applied to create mutants. Mutation based testing are most commonly used to individual statements for unit level testing and integration testing for classes. Mutation based testing is performed by creating alternative implementation which is known as mutant. The program which is under test is known as ground string. Ground strings are modified by the operator to create mutant program. Mutants are used to find tests. Testers are used to develop powerful test sets by effective selection of the mutation operator. When the output of a mutant program is not same like the correct output, then the mutant is considered as dead mutant and test case should kill that mutant. If the test case cannot kill at least one mutant is considered as ineffective and should be eliminated. Choosing test process for mutation is not so easy. Mutation analysis can be considered as a way to measure the quality of the test cases. In complete testing process, the tester should carefully examine the outputs of the test cases and the program should be fixed according the faults. The test case that detects the fault in a program also can kill a set of mutants (Amman & Offutt, 2008, chapter 5).

In section 4, related traditional techniques for the proposed methods are determined from the structural presentation of the testing method, which is mentioned in the first paragraph of each description of the testing methods. From table 2, we see that most of the testing methods for aspect-oriented software support graph based coverage criteria for the fault detection. None of the testing approach chose input space partitioning method because it seems that it is not possible to target these types of faults based on the input domain. Few methods use logic based coverage and all mutation based testing methods are mainly known as syntax based testing.

Therefore, from table 2, we can get a general idea about the testing methods for aspect-oriented software, which traditional techniques they are related. There are some testers those who are very efficient in graph based testing and some are very efficient in mutation testing. From the table 2, testers can get an option to choose a testing method according their efficiency. But still it is a matter that how much each of the testing methods can cover the faults while testing. Two of the testing methods are based on JUnit framework for unit testing (e.g., Lopes & Ngo, 2005 and Zhao & Alexander, 2007). Relation between traditional technique and testing method for aspect-oriented software is not specified for these two testing methods.

## 5.3 Goal of the proposed testing methods

Different test methods have different goals to achieve. There are two types of test goals; (i) which special portion of the aspect-oriented software is targeted by the methods and (ii) what development phase (e.g., unit testing, integration testing, system testing etc.) the methods apply to. Aspect-oriented program can be divided into two parts, for example the base program and the aspects. Aspects contain two parts: pointcut and advice. Through the weaving process aspects are composed with the base program. Some testing methods only target the pointcut expressions; some are intended to test the relations between class and aspects. On the other hand, some methods are based on traditional unit testing and some are based on traditional integration testing. Therefore, to specify the goals of the testing methods, at first, the common goals are specified, which are stated below.

1. **To reuse test cases of the base program:** An aspect-oriented software can be divided into two parts: one is referred as base program or core concern which consists traditional modularization units (e.g. classes, functions) and other consisting of aspects encapsulating the crosscutting concerns which is referred as secondary concern. Therefore, while testing aspect-oriented software, it is also important to test the base program and if the test cases for base program can be reused to test aspect, then it would be cost effective. If it is not possible to reuse the test cases, then some modification is required.

2. **To test pointcut expression:** We already know that pointcut is a part of aspect which is a statement that defines where the aspect will be woven into the program. It is very much mandatory to have proper pointcut expression for the proper weaving process. Therefore, to test pointcut expression is very important for aspect-oriented software. Some testing methods only test the pointcuts expressions and some does not target to test pointcut expression completely.

3. **To test aspects with related classes:** In aspect-oriented programming aspects are related with those classes whose behavior may be affected by one or more aspects. Therefore, it is impractical to test a class or an aspect in isolation. An aspect should be tested with those methods whose behavior may be affected by the aspect's advice and

**Table 3**: Goal of the proposed testing methods ("×"= targeted goal by the proposed testing method).

| Goal / Testing Methods | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Data flow based unit testing (Zhao, 2003) | | | × | × | | | |
| Aspect flow graph for testing (Xu et al., 2004) | × | | × | | × | × | |
| Inter-procedural Aspect Control Flow Graph based testing (Bernardi et al., 2007) | | × | × | | × | | |
| State-Based Approach (Xu et al., 2004) | | × | × | | × | × | |
| State-based incremental testing of aspect-oriented programs (Xu & Xu, 2006) | × | × | × | | × | × | × |
| Testing aspect-oriented programs with finite state machines (Xu et al., 2010) | × | × | × | | × | × | |
| A Model-Based Approach to Test Generation (Xu and Xu, 2005) | | | × | | × | × | |
| Testing aspect-oriented programs with uml design models. (Xu et al., 2008) | × | × | × | | × | × | |
| Testing AOS using uml activity diagram (Kaur & Grag, 2012) | × | × | × | | × | × | |
| Mutation testing of pointcuts (Anbalgan & Xie, 2006) | | × | | × | | | |
| Automated generation of pointcut mutants (Anbalagan & Xie, 2008) | | × | | × | | | |
| Mutation testing for aspect-oriented programs (Ferrari et al., 2008) | × | × | | × | | | |
| AjMutator: A Tool For the mutation analysis (Delamare et al., 2009) | | × | | × | | | |
| Aspect-oriented programming with model checking (Ubayashi & Tamai 2002) | | | × | × | | | |
| Towards a Practical Approach to Test Aspect-Oriented Software (Zhaou et al., 2004) | × | | × | × | × | × | |
| Unit-testing aspectual behavior (Lopes & Ngo, 2005) | | | × | × | | | |
| Testing as object-oriented programs (Zhao & Alexander, 2007) | | | × | × | | | |
| On the applicability of Random Testing (Parizi et al., 2009) | × | × | × | × | × | | |

class also should be tested together with those pieces of advice that may affect its behavior

4 **To conduct unit testing:** Unit testing verifies the detailed design for the unit has been correctly implemented by testing each unit (basic component) of a program (H.Zhu et al., 1997). Which testing methods for aspect-oriented software are targeting to conduct unit testing is identified here.

**5** **To conduct integration testing:** Some of the proposed testing approaches targeting interaction among the components, thus the methods targeting to conduct integration testing. Sometimes, unit testing is unable to detect faults, which are occurred for the interaction among the components. Therefore, integration testing should follow the unit testing.

**6** **To conduct system testing:** System testing indicates that the System is tested as a whole to check if the system meets functional requirement or not. System testing reflects the testing of behavior of a complete and fully integrated software product based on the software requirements. System testing is important to validate and verify both the application architecture and business requirements.

7 **To conduct traditional regression testing**: If there is any small change in any part of the program, it may cause problem in distinct part of the system. Regression testing is conducted to find out this kind of problem. Therefore, regression testing is known as retesting of the modified program. Some of the proposed testing methods are quite similar to traditional regression testing.

In section 4, goal for the proposed methods are determined from the flow of execution of the testing methods, which is described in the first paragraph of each description of the testing methods. Table 3 presents an overview about the goal of the proposed testing methods which helps to get an idea about the testing approaches at a glance. Among the testing methods, eight of them target to reuse the test cases for the base program. Most of the methods target to test pointcut expression and interactions among aspects and related classes. Eleven test approaches target to test pointcut expression and fourteen test approaches target to test aspects and related classes. If any tester wants to reuse the test cases for base program or intend to test pointcut expressions or intend to test interrelation between aspects and class, he/she can chose the testing method from the table 3. This is also applicable while choosing unit testing, integration testing, system testing and traditional regression testing. The researchers also can get the idea about the characteristics of the proposed methods from table 3. Among the eighteen test approaches ten of them conduct unit testing, ten of them conduct integration testing and eight of them perform system testing. Only one of the testing approaches is compared with traditional regression testing.

# 6 Syntheses of the testing methods

The analysis is incomplete without a synthesis. Syntheses reflect a combination of two or more entities that together form something new. That is, to check for patterns between different perspectives (addressed fault types, related tradition techniques and goal of the proposed testing method). Different perspectives are cross-checked each other. Three pair wise combination of criteria are presented in this chapter: 1) fault type vs. approach, 2) fault type vs. goal, 3) approach vs. goal. Each pair wise combination is presented in two tables. First table for each combination is mainly a two dimensional table. It means that the table represents information that is mainly two dimensional. In the second table for each combination, information is presented with explicit synthesis in large table. Each of the testing methods is specified by a letter, which is stated below.

A. Aspect-oriented programming with model checking(Ubayashi & Tamai, 2002)
B. Data flow based unit testing (Zhao 2003)
C. Aspect flow graph for testing (Xu et al., 2004)
D. State-Based Approach (Xu et al., 2004)
E. Towards a practical approach to Test Aspect-Oriented Software (Zhaou et al., 2004)
F. A Model-Based Approach to Test Generation (Xu and Xu, 2005)
G. Unit-testing aspectual behavior (Lopes & Ngo, 2005)
H. State-based incremental testing of aspect-oriented programs (Xu & Xu, 2006)
I. Mutation testing of pointcuts (Anbalgan & Xie, 2006)
J. Inter-procedural Aspect Control Flow Graph based testing (Bernardi et al., 2007)
K. Testing as object-oriented programs (Zhao & Alexander, 2007)
L. Testing aspect-oriented programs with uml design models. (Xu et al., 2008)
M. Automated generation of pointcut mutants (Anbalagan & Xie, 2008)
N. Mutation testing for aspect-oriented programs (Ferrari et al., 2008)
O. AjMutator: A Tool For the mutation analysis (Delamare et al., 2009)
P. On the applicability of Random Testing (Parizi et al., 2009)
Q. Testing aspect-oriented programs with finite state machines (Xu et al., 2010)
R. Testing AOS using uml activity diagram (Kaur & Grag, 2012)

## 6.1 Pair wise analyses between Fault type and approach

Rows of the table 4(A) and 4(B) represent approaches of the testing methods and columns represent the fault types that the proposed testing approaches target. The following table presents syntheses, which reflect the relation between fault type and approach of the proposed testing methods. Table 4(B) explicit all information in a large scale that presents the relations between the fault type, approach and testing method in a more clear view. Different coloring schemes are used in table 4(A) and table 4(B) to get a more identical view of the testing methods. The fields of the table 4(A) and 4 (B) are marked with blue color that represent for unknown testing approach or fault type that is not targeted by any testing method. On the contrary, the fields with green color represent for the testing methods that target any of the mentioned approaches and also target any of the presented fault types. The number of the fault type is same as the number of fault type that is mentioned in section 5.1. Each decision for the placement of each testing method in table 4(A) and 4(B) is derived from table 1 and 2 of section 5.

**Table 4(A):** Pair wise analysis between fault type and approach.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Not target any fault type |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Graph-based | D,H,J,L, Q,R | L,Q, R | | B,D, J | J | C,J | J | Q | | H,L, Q,R | A,E,F |
| Logic-based | D,J,L | L | | D,J | J | J | J | | | L | |
| Input space partitioning | | | | | | | | | | | |
| Syntax testing | I,M,N,O ,Q | Q | | | | | | N,Q | | N,Q | |
| Not applicable/ unknown | | | | | | | | | | | G,K,P |

**Findings:**

I) It is interesting that the logic based approaches miss to target fault type 8 (Failure due to inter-type declarations).

II) Concerning mutation testing, most are on unit-level. Only Q is on integration and system testing level.

III) Testing approach Q (Testing aspect-oriented programs with finite state machines) and testing approach L (Testing aspect-oriented programs with uml design models) targeting the same faults while testing approach Q (Testing aspect-oriented programs with finite state machines) and testing approach J (Inter-procedural Aspect Control Flow Graph based testing) targeting the different fault type.

**Discussions:**

In case of inter type declaration; static cross-cutting modifies the base code, which may produce ripple effects in the control flow. Logic based approach is quite difficult to target faults here. In this circumstances, if we consider Boolean operation x instance of y produce a true value and the dynamic type of the object x is a subtype of y. If an aspect changes the inheritance relation of x then the previous Boolean operator may return a different value, which is altering the control flow.

In case of Syntax based test approaches, testing approach Q (Testing aspect-oriented programs with finite state machines) and N (Mutation testing for aspect-oriented programs) targets common fault types while Q target one more fault type than N. Therefore, instead of N, Q can be a better choice to test aspect-oriented software.

**Table 4(B):** Pair wise analysis between fault type and approach in details.

| **Method** | **Approach** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Not target any fault type |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | Graph-based | | | | | | | | | | | A |
| | Logic-based | | | | | | | | | | | |
| | Input space partitioning | | | | | | | | | | | |
| | Syntax testing | | | | | | | | | | | |
| | Not applicable/unknown | | | | | | | | | | | |
| B | Graph-based | | | | B | | | | | | | |
| | Logic-based | | | | | | | | | | | |
| | Input space partitioning | | | | | | | | | | | |
| | Syntax testing | | | | | | | | | | | |
| | Not applicable/unknown | | | | | | | | | | | |
| C | Graph-based | | | | | | C | | | | | |
| | Logic-based | | | | | | | | | | | |
| | Input space | | | | | | | | | | | |

| Method | Approach | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Not target any fault type |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | partitioning | | | | | | | | | | | |
| | Syntax testing | | | | | | | | | | | |
| | Not applicable/unknown | | | | | | | | | | | |
| D | Graph-based | D | | | D | | | | | | | |
| | Logic-based | D | | | D | | | | | | | |
| | Input space partitioning | | | | | | | | | | | |
| | Syntax testing | | | | | | | | | | | |
| | Not applicable/unknown | | | | | | | | | | | |
| E | Graph-based | | | | | | | | | | | E |
| | Logic-based | | | | | | | | | | | |
| | Input space partitioning | | | | | | | | | | | |
| | Syntax testing | | | | | | | | | | | |
| | Not applicable/unknown | | | | | | | | | | | |
| F | Graph-based | | | | | | | | | | | F |
| | Logic-based | | | | | | | | | | | |
| | Input space partitioning | | | | | | | | | | | |
| | Syntax testing | | | | | | | | | | | |
| | Not applicable/unknown | | | | | | | | | | | |
| G | Graph-based | | | | | | | | | | | |
| | Logic-based | | | | | | | | | | | |
| | Input space partitioning | | | | | | | | | | | |
| | Syntax testing | | | | | | | | | | | |
| | Not applicable/unknown | | | | | | | | | | | G |
| H | Graph-based | H | | | | | | | | | H | |
| | Logic-based | | | | | | | | | | | |
| | Input space partitioning | | | | | | | | | | | |
| | Syntax testing | | | | | | | | | | | |
| | Not applicable/unknown | | | | | | | | | | | |
| I | Graph-based | | | | | | | | | | | |
| | Logic-based | | | | | | | | | | | |
| | Input space partitioning | | | | | | | | | | | |
| | Syntax testing | I | | | | | | | | | | |
| | Not | | | | | | | | | | | |

| **Method** | **Approach** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Not target any fault type |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | applicable/unknown | | | | | | | | | | | |
| J | Graph-based | J | | | J | J | J | J | | | | |
| | Logic-based | J | | | J | J | J | J | | | | |
| | Input space partitioning | | | | | | | | | | | |
| | Syntax testing | | | | | | | | | | | |
| | Not applicable/unknown | | | | | | | | | | | |
| K | Graph-based | | | | | | | | | | | |
| | Logic-based | | | | | | | | | | | |
| | Input space partitioning | | | | | | | | | | | |
| | Syntax testing | | | | | | | | | | | |
| | Not applicable/unknown | | | | | | | | | | | K |
| L | Graph-based | L | L | | | | | | | | L | |
| | Logic-based | L | L | | | | | | | | L | |
| | Input space partitioning | | | | | | | | | | | |
| | Syntax testing | | | | | | | | | | | |
| | Not applicable/unknown | | | | | | | | | | | |
| M | Graph-based | | | | | | | | | | | |
| | Logic-based | | | | | | | | | | | |
| | Input space partitioning | | | | | | | | | | | |
| | Syntax testing | M | | | | | | | | | | |
| | Not applicable/unknown | | | | | | | | | | | |
| N | Graph-based | | | | | | | | | | | |
| | Logic-based | | | | | | | | | | | |
| | Input space partitioning | | | | | | | | | | | |
| | Syntax testing | N | | | | | | | N | | N | |
| | Not applicable/unknown | | | | | | | | | | | |
| O | Graph-based | | | | | | | | | | | |
| | Logic-based | | | | | | | | | | | |
| | Input space partitioning | | | | | | | | | | | |
| | Syntax testing | O | | | | | | | | | | |
| | Not applicable/unknown | | | | | | | | | | | |
| p | Graph-based | | | | | | | | | | | |
| | Logic-based | | | | | | | | | | | |

| Method | Approach | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Not target any fault type |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Input space partitioning | | | | | | | | | | | |
| | Syntax testing | | | | | | | | | | | |
| | Not applicable/unknown | | | | | | | | | | | P |
| Q | Graph-based | Q | Q | | | | | | Q | | Q | |
| | Logic-based | | | | | | | | | | | |
| | Input space partitioning | | | | | | | | | | | |
| | Syntax testing | Q | Q | | | | | | Q | | Q | |
| | Not applicable/unknown | | | | | | | | | | | |
| R | Graph-based | R | R | | | | | | | | R | |
| | Logic-based | | | | | | | | | | | |
| | Input space partitioning | | | | | | | | | | | |
| | Syntax testing | | | | | | | | | | | |
| | Not applicable/unknown | | | | | | | | | | | |

**Findings:**

I) In the above table, every field is considered as two dimensional field which represents the testing approach and targeting fault type. The testing method J (Inter-procedural Aspect Control Flow Graph based testing) covers most of the fields than any other testing approaches.

II) Fault types: 3 (Failure to establish expected post-conditions) and 9 (Incorrect changes in polymorphic calls) are targeted by no other testing methods as well as no other testing approaches. Except fault type 3 and 9, all other fault types are targeted by graph based testing approach. Logic based testing approach cannot target fault type 8 (Failure due to inter-type declarations) also.

III) Syntax testing target fault type 1, 2, 8 and 10. Testing method Q performs better among all syntax based testing approaches.

**Discussions:**

The testing method J (Inter-procedural Aspect Control Flow Graph based testing) targets the highest number of fault type and it also based on graph based testing approach and logic based testing approach. Therefore it is obvious that the testing method J will cover highest number of fields in the above table.

It is interesting that the logic based approaches miss to target fault type 8 (Failure due to inter-type declarations). In case of inter type declaration; static cross-cutting modifies the base code,

which may produce ripple effects in the control flow. Logic based approach is quite difficult to target faults here. In this circumstances if we consider Boolean operation x instance of y produce a true value and the dynamic type of the object x is a subtype of y. If an aspect changes the inheritance relation of x then the previous Boolean operator may return a different value, which is altering the control flow.

## 6.2 Pair wise analysis between Fault type and goal

In Table 5(A) and 5(B), there are two dimensional information is presented. According the column, each testing method shows the targeting fault type and according the row, the testing method address a specific goal that it targets. Each column of the table represents a fault type and each row represents a goal of the testing method. The number of the fault type is same as the presented fault type in section 5.1. Different coloring schemes are used in table 6(A) and 6(B). Goal of the testing methods can be divided into three parts: using (reuse + regression) (1+7), aspect-oriented (2+3) and levels of testing (4-6). Goal 1 and 7 are marked with green color, 2 and 3 are marked with purple color and goal 4, 5, 6 are marked with red color. The number of the fault type is same as the number of fault type that is mentioned in section 5.1. Each decision for the placement of each testing method in table 5(A) and 5(B) is derived from table 1 and 3 of section 5. If we mark them with colors, the result can be as follows:

**Table 5(A):** Pair wise analysis between fault type and goal.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Not target any fault type |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1.To reuse test cases of the base program | H,L,N,Q,R | L,Q,R | | | | C | | N,Q | | H,L,N,Q,R | E,P |
| 2.To test pointcut expression | D,H,I,J,L,M,N,O,Q, R | L,Q,R | | D,J | J | J | J | N,Q | | H,L,N,Q,R | P |
| 3.To test aspects with related classes: | D,H,J,L,Q,R | L,Q,R | | B,D,J | J | C,J | J | Q | | H,L,Q,R | A,E,F,G,K,P |
| 4.To conduct unit testing | I,K,M,N,O | | | B | | | | N | | N | A,E,G,P |
| 5.To conduct integration testing | D,H,J,L,Q,R | L,Q,R | | D,J | J | C,J | J | Q | | H,L,Q,R | E,F,P |
| 6.To conduct system testing | D,H,L,Q,R | L,Q,R | | D | | C | | Q | | H,L,Q,R | E,F |
| 7.To conduct traditional regression testing | H | | | | | | | | | H | |

**Findings:**

I) Few faults are targeted on unit level testing.
II) Most of the faults are targeted while testing aspect with related classes and integration testing.

**Discussions:**

Testing of aspect oriented-software mainly targets to test the interaction between the aspects and related classes, which is a kind of integration testing. From the above table, we find that most of the testing methods target to test aspect with related classes.

**Table 5(B):** Pair wise analysis between fault type and goal in details.

| **Method** | **Approach** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Not target any fault type |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | 1.To reuse test cases of the base program | | | | | | | | | | | |
| | 2.To test pointcut expression | | | | | | | | | | | |
| | 3.To test aspects with related classes: | | | | | | | | | | | A |
| | 4.To conduct unit testing | | | | | | | | | | | A |
| | 5.To conduct integration testing | | | | | | | | | | | |
| | 6.To conduct system testing | | | | | | | | | | | |
| | 7.To conduct traditional regression testing | | | | | | | | | | | |
| B | 1.To reuse test cases of the base program | | | | | | | | | | | |
| | 2.To test pointcut expression | | | | | | | | | | | |
| | 3.To test aspects with related classes: | | | | B | | | | | | | |
| | 4.To conduct unit testing | | | | B | | | | | | | |

| **Method** | **Approach** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Not target any fault type |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 5.To conduct integration testing | | | | | | | | | | | |
| | 6.To conduct system testing | | | | | | | | | | | |
| | 7.To conduct traditional regression testing | | | | | | | | | | | |
| C | 1.To reuse test cases of the base program | | | | | | C | | | | | |
| | 2.To test pointcut expression | | | | | | | | | | | |
| | 3.To test aspects with related classes: | | | | | | C | | | | | |
| | 4.To conduct unit testing | | | | | | | | | | | |
| | 5.To conduct integration testing | | | | | | C | | | | | |
| | 6.To conduct system testing | | | | | | C | | | | | |
| | 7.To conduct traditional regression testing | | | | | | | | | | | |
| D | 1.To reuse test cases of the base program | | | | | | | | | | | |
| | 2.To test pointcut expression | D | | | D | | | | | | | |
| | 3.To test aspects with related classes: | D | | | D | | | | | | | |
| | 4.To conduct unit testing | | | | | | | | | | | |
| | 5.To conduct integration testing | D | | | D | | | | | | | |
| | 6.To conduct system testing | D | | | D | | | | | | | |
| | 7.To conduct traditional regression testing | | | | | | | | | | | |
| E | 1.To reuse test cases of the base program | | | | | | | | | | | E |
| | 2.To test pointcut expression | | | | | | | | | | | |

| **Method** | **Approach** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Not target any fault type |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 3.To test aspects with related classes: | | | | | | | | | | | E |
| | 4.To conduct unit testing | | | | | | | | | | | E |
| | 5.To conduct integration testing | | | | | | | | | | | E |
| | 6.To conduct system testing | | | | | | | | | | | E |
| | 7.To conduct traditional regression testing | | | | | | | | | | | |
| F | 1.To reuse test cases of the base program | | | | | | | | | | | |
| | 2.To test pointcut expression | | | | | | | | | | | |
| | 3.To test aspects with related classes: | | | | | | | | | | | F |
| | 4.To conduct unit testing | | | | | | | | | | | |
| | 5.To conduct integration testing | | | | | | | | | | | F |
| | 6.To conduct system testing | | | | | | | | | | | F |
| | 7.To conduct traditional regression testing | | | | | | | | | | | |
| G | 1.To reuse test cases of the base program | | | | | | | | | | | |
| | 2.To test pointcut expression | | | | | | | | | | | |
| | 3.To test aspects with related classes: | | | | | | | | | | | G |
| | 4.To conduct unit testing | | | | | | | | | | | G |
| | 5.To conduct integration testing | | | | | | | | | | | |
| | 6.To conduct system testing | | | | | | | | | | | |
| | 7.To conduct traditional regression testing | | | | | | | | | | | |

| **Method** | **Approach** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Not target any fault type |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | 1.To reuse test cases of the base program | H | | | | | | | | | H | |
| | 2.To test pointcut expression | H | | | | | | | | | H | |
| | 3.To test aspects with related classes: | H | | | | | | | | | H | |
| | 4.To conduct unit testing | | | | | | | | | | | |
| | 5.To conduct integration testing | H | | | | | | | | | H | |
| | 6.To conduct system testing | H | | | | | | | | | H | |
| | 7.To conduct traditional regression testing | H | | | | | | | | | H | |
| I | 1.To reuse test cases of the base program | | | | | | | | | | | |
| | 2.To test pointcut expression | I | | | | | | | | | | |
| | 3.To test aspects with related classes: | | | | | | | | | | | |
| | 4.To conduct unit testing | I | | | | | | | | | | |
| | 5.To conduct integration testing | | | | | | | | | | | |
| | 6.To conduct system testing | | | | | | | | | | | |
| | 7.To conduct traditional regression testing | | | | | | | | | | | |
| J | 1.To reuse test cases of the base program | | | | | | | | | | | |
| | 2.To test pointcut expression | J | | | J | J | J | J | | | | |
| | 3.To test aspects with related classes: | J | | | J | J | J | J | | | | |
| | 4.To conduct unit testing | | | | | | | | | | | |
| | 5.To conduct integration testing | J | | | J | J | J | J | | | | |

| **Method** | **Approach** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Not target any fault type |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 6.To conduct system testing | | | | | | | | | | | |
| | 7.To conduct traditional regression testing | | | | | | | | | | | |
| K | 1.To reuse test cases of the base program | | | | | | | | | | | |
| | 2.To test pointcut expression | | | | | | | | | | | |
| | 3.To test aspects with related classes: | | | | | | | | | | | K |
| | 4.To conduct unit testing | | | | | | | | | | | K |
| | 5.To conduct integration testing | | | | | | | | | | | |
| | 6.To conduct system testing | | | | | | | | | | | |
| | 7.To conduct traditional regression testing | | | | | | | | | | | |
| L | 1.To reuse test cases of the base program | L | L | | | | | | | | L | |
| | 2.To test pointcut expression | L | L | | | | | | | | L | |
| | 3.To test aspects with related classes: | L | L | | | | | | | | L | |
| | 4.To conduct unit testing | | | | | | | | | | | |
| | 5.To conduct integration testing | L | L | | | | | | | | L | |
| | 6.To conduct system testing | L | L | | | | | | | | L | |
| | 7.To conduct traditional regression testing | | | | | | | | | | | |
| M | 1.To reuse test cases of the base program | | | | | | | | | | | |
| | 2.To test pointcut expression | M | | | | | | | | | | |
| | 3.To test aspects with related | | | | | | | | | | | |

| Method | Approach | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Not target any fault type |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | classes: | | | | | | | | | | | |
| | 4.To conduct unit testing | M | | | | | | | | | | |
| | 5.To conduct integration testing | | | | | | | | | | | |
| | 6.To conduct system testing | | | | | | | | | | | |
| | 7.To conduct traditional regression testing | | | | | | | | | | | |
| N | 1.To reuse test cases of the base program | N | | | | | | | N | | N | |
| | 2.To test pointcut expression | N | | | | | | | N | | N | |
| | 3.To test aspects with related classes: | | | | | | | | | | | |
| | 4.To conduct unit testing | N | | | | | | | N | | N | |
| | 5.To conduct integration testing | | | | | | | | | | | |
| | 6.To conduct system testing | | | | | | | | | | | |
| | 7.To conduct traditional regression testing | | | | | | | | | | | |
| O | 1.To reuse test cases of the base program | | | | | | | | | | | |
| | 2.To test pointcut expression | O | | | | | | | | | | |
| | 3.To test aspects with related classes: | | | | | | | | | | | |
| | 4.To conduct unit testing | O | | | | | | | | | | |
| | 5.To conduct integration testing | | | | | | | | | | | |
| | 6.To conduct system testing | | | | | | | | | | | |
| | 7.To conduct traditional regression testing | | | | | | | | | | | |
| P | 1.To reuse test | | | | | | | | | | | P |

| **Method** | **Approach** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Not target any fault type |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | cases of the base program | | | | | | | | | | | |
| | 2.To test pointcut expression | | | | | | | | | | | P |
| | 3.To test aspects with related classes: | | | | | | | | | | | P |
| | 4.To conduct unit testing | | | | | | | | | | | P |
| | 5.To conduct integration testing | | | | | | | | | | | P |
| | 6.To conduct system testing | | | | | | | | | | | |
| | 7.To conduct traditional regression testing | | | | | | | | | | | |
| Q | 1.To reuse test cases of the base program | Q | Q | | | | | | Q | | Q | |
| | 2.To test pointcut expression | Q | Q | | | | | | Q | | Q | |
| | 3.To test aspects with related classes: | Q | Q | | | | | | | | Q | |
| | 4.To conduct unit testing | | | | | | | | | | | |
| | 5.To conduct integration testing | Q | Q | | | | | | Q | | Q | |
| | 6.To conduct system testing | Q | Q | | | | | | Q | | Q | |
| | 7.To conduct traditional regression testing | | | | | | | | | | | |
| R | 1.To reuse test cases of the base program | R | R | | | | | | | | R | |
| | 2.To test pointcut expression | R | R | | | | | | | | R | |
| | 3.To test aspects with related classes: | R | R | | | | | | | | R | |
| | 4.To conduct unit testing | | | | | | | | | | | |
| | 5.To conduct integration testing | R | R | | | | | | | | R | |

| Method | Approach | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Not target any fault type |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 6.To conduct system testing | R | R | | | | | | | | R | |
| | 7.To conduct traditional regression testing | | | | | | | | | | | |

**Findings:**

I) In table 5(B), every field is considered as two dimensional field which represents the goal and targeting fault type of the testing methods. The testing method Q (Testing aspect-oriented programs with finite state machines) covers most of the fields than any other testing approaches.

II) Few faults are targeted on unit level testing.

III) Most of the faults are targeted while testing aspect with related classes and integration testing.

**Discussions:**

Table 5(B), reveals that the testing method H (State-based incremental testing of aspect-oriented programs) targets the highest number of the goals but targets only two fault types. On the contrary, the testing method J (Inter-procedural Aspect Control Flow Graph based testing) targets highest number of fault types among all testing methods but targets only two goals. Therefore, it seems that a testing method can target a single fault type satisfying different goals and vice versa. The testing method Q (Testing aspect-oriented programs with finite state machines) covers most of the fields as it targets four fault types and 5 goals.

## 6.3 Pair wise analysis between Approach and Goal

In table 6(A) and 6(B), testing methods are presented according the related approach and goal. Each column represents goal for the testing method and each row represents approach that the testing method is based on. The goals are numbered according to the order in table 5 (A) and 5(B). Each decision for the placement of each testing method in table 6(A) and 6(B) is derived from table 2 and 3 of section 5. Different coloring schemes are also used here like the previous tables. Three parts of goals are presented by three different colors: using (reuse + regression) (1+7), aspect-oriented (2+3) and levels of testing (4-6). Goal 1 and 7 are marked with green color, 2 and 3 are marked with purple color and goal 4, 5, 6 are marked with red color. If we mark them with colors, the result can be as follows:

**Table 6(A):** Pair wise analysis between approach and goal.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Graph-based | C, E, H, L, Q, R | D, H, J, L, Q, R | A, B, C, D, E, F, H, J, L, Q, R | A, B, E | C, D, E, F, H, J, L, Q, R | C, D, E, F, H, L, Q, R | H |
| Logic-based | L | D, J, L | D, J, L | | D, J, L | D, L | |
| Input space partitioning | | | | | | | |
| Syntax testing | N, Q | I, M, N, O, Q | Q | I, M, N, O | Q | Q | |
| Not applicable/unknown | P | P | G, K, P | G, K, P | P | | |

**Findings:**

I) Concerning reuse of test cases, most are in graph-based testing for some reason.
II) Concerning mutation testing, most are on unit-level. Only testing method Q supports integration and system testing.
III) No logic-based is on unit level testing.

**Discussions:**

Logic based methods are not available in unit level testing in most of the cases. In the weave time, pointcut designator selects a method's join points to be woven. Most of the cases it requires proper conditions or logics while the selection is determined. Logic coverage is mainly used to target this type of interactions.

**Table 6(B):** Pair wise analysis between approach and goal in details.

| Method | Approach | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | Graph-based | | | A | A | | | |
| | Logic-based | | | | | | | |
| | Input space partitioning | | | | | | | |
| | Syntax testing | | | | | | | |
| | Not applicable/unknown | | | | | | | |
| B | Graph-based | | | B | B | | | |
| | Logic-based | | | | | | | |
| | Input space partitioning | | | | | | | |
| | Syntax testing | | | | | | | |
| | Not applicable/unknown | | | | | | | |
| C | Graph-based | C | | C | | C | C | |
| | Logic-based | | | | | | | |
| | Input space partitioning | | | | | | | |
| | Syntax testing | | | | | | | |
| | Not applicable/unknown | | | | | | | |
| D | Graph-based | | D | D | | D | D | |
| | Logic-based | | D | D | | D | D | |
| | Input space partitioning | | | | | | | |
| | Syntax testing | | | | | | | |
| | Not applicable/unknown | | | | | | | |
| E | Graph-based | E | | E | E | E | E | |
| | Logic-based | | | | | | | |
| | Input space partitioning | | | | | | | |
| | Syntax testing | | | | | | | |
| | Not applicable/unknown | | | | | | | |
| F | Graph-based | | | F | | F | F | |
| | Logic-based | | | | | | | |
| | Input space partitioning | | | | | | | |

| Method | Approach | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | Syntax testing | | | | | | | |
| | Not applicable/unknown | | | | | | | |
| G | Graph-based | | | | | | | |
| | Logic-based | | | | | | | |
| | Input space partitioning | | | | | | | |
| | Syntax testing | | | | | | | |
| | Not applicable/unknown | | | G | G | | | |
| H | Graph-based | H | H | H | | H | H | H |
| | Logic-based | | | | | | | |
| | Input space partitioning | | | | | | | |
| | Syntax testing | | | | | | | |
| | Not applicable/unknown | | | | | | | |
| I | Graph-based | | | | | | | |
| | Logic-based | | | | | | | |
| | Input space partitioning | | | | | | | |
| | Syntax testing | | I | | I | | | |
| | Not applicable/unknown | | | | | | | |
| J | Graph-based | | J | J | | J | | |
| | Logic-based | | J | J | | J | | |
| | Input space partitioning | | | | | | | |
| | Syntax testing | | | | | | | |
| | Not applicable/unknown | | | | | | | |
| K | Graph-based | | | | | | | |
| | Logic-based | | | | | | | |
| | Input space partitioning | | | | | | | |
| | Syntax testing | | | | | | | |
| | Not applicable/unknown | | | K | K | | | |
| L | Graph-based | L | L | L | | L | L | |
| | Logic-based | L | L | L | | L | L | |
| | Input space partitioning | | | | | | | |

| Method | Approach | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | Syntax testing | | | | | | | |
| | Not applicable/unknown | | | | | | | |
| M | Graph-based | | | | | | | |
| | Logic-based | | | | | | | |
| | Input space partitioning | | | | | | | |
| | Syntax testing | | M | | M | | | |
| | Not applicable/unknown | | | | | | | |
| N | Graph-based | | | | | | | |
| | Logic-based | | | | | | | |
| | Input space partitioning | | | | | | | |
| | Syntax testing | N | N | | N | | | |
| | Not applicable/unknown | | | | | | | |
| O | Graph-based | | | | | | | |
| | Logic-based | | | | | | | |
| | Input space partitioning | | | | | | | |
| | Syntax testing | | O | | O | | | |
| | Not applicable/unknown | | | | | | | |
| P | Graph-based | | | | | | | |
| | Logic-based | | | | | | | |
| | Input space partitioning | | | | | | | |
| | Syntax testing | | | | | | | |
| | Not applicable/unknown | P | P | P | P | P | | |
| Q | Graph-based | Q | Q | Q | | Q | Q | |
| | Logic-based | | | | | | | |
| | Input space partitioning | | | | | | | |
| | Syntax testing | Q | Q | Q | | Q | Q | |
| | Not applicable/unknown | | | | | | | |
| R | Graph-based | R | R | R | | R | R | |
| | Logic-based | | | | | | | |
| | Input space partitioning | | | | | | | |

| Method | Approach | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | Syntax testing | | | | | | | |
| | Not applicable/unknown | | | | | | | |

**Findings:**

I) Testing method Q (Testing aspect-oriented programs with finite state machines) and testing method L (Testing aspect-oriented programs with uml design models) covers highest number of two dimensional fields.

II) Most of the testing methods that support reuse of test cases are in graph-based testing for some reason. Most of the mutation based testing methods are on unit-level. Only testing method Q supports integration and system testing.

III) No logic-based is on unit level testing.

**Discussions:**

From the table 7(B), we can tell that only D, L and Q are based on different groups. They address both point cut expressions and related classes at integration and system level as well as reuse test cases. Only H is considering regression, but a lot of others are considering reuse. We can also tell that A, B, G, K, M, O are rather limited as they are able to cover few fields compared to other methods.

The findings between table 7(A) and 7(B) are same. Therefore, discussions are more or less same for both the tables. Logic based testing methods does not support unit level testing. In most of the cases, logic is applied to target the interactions between aspects and functions.

# 7 Discussions

This section presents an overall discussion about the current dissertation. The discussion is based on the results of the current dissertation. It also addresses validity of the results as well as ethics. The following results are found by comparing and contrasting the methods.

1. Section 5.1, presents a fault model of aspect-oriented software. None of the proposed testing methods target two fault types: failure to establish expected post-conditions and incorrect changes in the polymorphic calls. Among all testing methods, Inter-procedural Aspect Control Flow Graph based testing method (Mario Luca Bernardi et al., 2007) targets highest number (5) of faults in the fault model.

2. Most of the proposed testing methods for aspect-oriented software are based on some traditional techniques or there are relations between the proposed testing methods and the traditional techniques. From the investigation, we find that most testing methods are based on graph based coverage, logic based coverage and mutation testing which is known as syntax-based testing. Comparatively, graph based testing techniques are applied more than the logic based coverage and mutation testing. Some testing methods use both graph based and logic based testing techniques. Mutation testing is effective to test pointcut descriptions of the aspects.

3. Some of the testing methods (e.g., especially the state based approaches) intend to reuse the test cases for the base program which reduce the testing cost and reduce the test cases also. Concerning reuse of test cases, most are in graph-based testing for some reason.

4. While investigating the proposed testing methods, we find that there is a common limitation among the testing methods for aspect oriented software. None of the testing method has all testing feature for testing aspect-oriented software. Therefore, it is recommended not to depend upon a single testing method while testing aspect-oriented software.

The first result reveals that none of the testing methods target two fault types (fault type 3 and 9) and Inter-procedural Aspect Control Flow Graph based testing method targets highest number of fault types. This result is found by comparing eighteen individual testing methods for aspect-oriented software. Comparison with a good number of testing methods makes the result more reliable. Table 1 shows the targeting fault type for each testing methods. Five of the testing methods do not target any of the specific fault type. But for other testing methods, targeting fault type is specified clearly in the related papers. Therefore, there are no confusions about the targeting fault types. From the second result we find that most of the testing methods are based on some traditional techniques or there are relations between traditional techniques and techniques for aspect-oriented software. At first, traditional techniques are specified clearly. Table 2 shows the relations between traditional techniques and testing methods for aspect-oriented software. Three of the testing methods do not match any criteria that we can determine that they are based on any traditional techniques. Therefore, it is defined that these three testing

methods do not support any traditional technique clearly. Other testing methods strongly support any criteria of the traditional techniques. Section 6 presents a synthesis. Synthesis reflects an experiment that reveals something new. Different perspectives are cross-checked in the synthesis part. Without this synthesis, the analysis would not complete.

There are three objectives to fulfill of this dissertation. The first objective is to survey test methods for aspect-oriented software. Section 4 presents the survey of the test methods, which met the first objective of the dissertation. The survey provides an idea about the general development of the testing methods for aspect-oriented software. The survey not only presents the testing methods but also describes that how the methods are used and which purpose they fulfill. (e.g., "graphs are used to derive input parameters that cover all paths meeting some criteria"). The second objective is to identify a set of criteria to categorize the testing approaches. In section 5, three criteria are identified as a basis of the analysis. These criteria help not only to categorize the testing approaches but also to perform and present an analysis about the testing approaches and thereby it met the second objective. The third objective is to analyze the testing approaches. In section 5 and 6, there are some interesting findings about the testing approaches of aspect-oriented software. The important findings are revealed by contrasting and combining results from three different studies that complete the overview over testing approaches and thereby the third objective is also met. As all three objectives are fulfilled, the overall aim of the thesis is reached.

The survey over testing approaches in the current dissertation presents more testing approaches than any other survey work over testing approaches of aspect-oriented software that is available. For the current survey, chosen databases are: 1) IEEE Xplore, 2) ACM Digital Library, and 3) Springer Link. Each of the databases contains a comprehensive archive of the organization's journals, magazines, and conference proceedings. Search terms are: 1) "Aspect-oriented software testing", 2) "Aspect-oriented software verification", 3) "Testing approaches for aspect-oriented software", 4) "Testing methods for aspect-oriented software". To select the papers from the proposed databases, at first, it is examined that the title is ok. Then, the abstract of the paper is read. If the abstract support the current research, then the paper is read. It is quite difficult to demand that the current survey does not miss any important article. There are also possibilities that an important article about the testing method of aspect-oriented software is not published in the above mentioned database. To overcome this limitation, all the references of the related papers are examined properly that the important journals are not missed. Therefore, it can be claimed that the analysis about the testing approaches for aspect-oriented software is more enrich than any other previous analysis. The survey over testing methods of aspect-oriented software shows that the information of analysis data is valid. Therefore, we can claim that the results of this dissertation is authenticate. From the above results of the current dissertation we can also decide that still it requires quite good advancement for the testing approach of aspect-oriented software. There are opportunities to do more to attain effective and complete testing approach for aspect-oriented software.

# 8 Related Work

In this section, this dissertation is compared with some other related work. The differences between the approaches and contents are discussed in the following section. At first, the related papers are presented and then the comparison between the related papers and current dissertation is summarized in a table.

Naqvi et al. (2005) conducted a survey for the testing methods of aspect-oriented software. The authors present an analysis of the testing strategies for aspect-oriented software. Problem definition of this paper is more or less similar like the current dissertation. In this paper, a state of art is presented. There are three testing strategies have been examined in this paper. The authors describe three testing methods: Data flow based unit testing of AOP, State based testing approach and Aspect flow graph based testing. The effectiveness of these testing methods is measured in terms of their fault finding abilities. Current dissertation surveyed eighteen testing methods for aspect-oriented software. The authors present a fault model with six fault types for the aspect-oriented program which is based on Alexander (2004). There are ten fault types in the fault model of the current dissertation. Among the three testing methods, the authors decide that the state based approach has the better potential of revealing faults, which makes the authors more interested to further investigate state based methods. According to the investigation in this thesis, Inter-procedural Aspect Control Flow Graph based testing method (Mario Luca Bernardi et al., 2007) targets highest number (5) of fault types. Naqvi et al. (2005) only present the relationship between the method and fault type. In the current dissertation, the testing approaches are discussed with respect to their relation with traditional techniques, and their goals. In this paper, authors emphasize the survey over testing methods and fault finding capabilities of three testing methods. On the contrary, the current dissertation not only checks the method and fault type but also presents and analyzes and synthesis over the testing methods for aspect-oriented software.

Parizi & Ghani (2007) presents a survey for the testing method of aspect-oriented software, which is extended from the previous survey (e.g., Syed Asad Ali Naqvi, 2005). The authors present more clear background on testing and aspect-oriented programming concepts. There are five testing methods are surveyed in this paper. In survey of the current dissertation contains eighteen testing methods for aspect-oriented software. This paper also presents the fault model with six fault types that are introduced by Alexander (2004). There are ten fault types in the fault model of the current dissertation. In this paper, the effectiveness of the proposed testing approaches is compared in terms of their fault finding abilities. But the experiment of fault finding capabilities of this paper does not differ from the previous paper (e.g., Syed Asad Ali Naqvi, 2005). In this paper, only three testing approaches target the fault model and other two cannot be determined. In this perspective, the contribution of this paper is similar like the paper of Syed Asad Ali Naqvi (2005). In the summary of the survey the authors claim that the state based approach is comparatively better than the other testing approaches. As a future work the authors plan to include more testing approaches. In this paper, authors conduct a survey and checks fault finding capabilities of the testing methods. Current dissertation not only checks the method and fault type but also checks the relation with traditional techniques and goals of the testing methods.

Kumer et al. (2009) conduct a survey over testing methods of aspect-oriented software. The authors present a summary of state of art at first, and then they mention the challenges of testing aspect-oriented software. The problem definition of this paper is to provide an analytical overview over current testing methods of aspect-oriented software, which is quite similar like the current dissertation. In this paper, authors enlarge the fault model incorporating three more fault types which are presented by Ceccato (2005). Therefore the number of fault types is nine in the fault model. Current dissertation adds one more fault type that is targeted by the testing methods in the recent articles. There are only four testing methods are surveyed in this paper. Current dissertation surveyed eighteen testing methods. Like other survey works, effectiveness of testing methods is evaluated by the fault finding capabilities. This paper finds that fault type 2, 3, 8 and 9 are not targeted by any testing method. On the contrary current dissertation finds that fault type 3 and 9 are not targeted. This is an advancement of the dissertation. But part of the result between this paper and current dissertation is same as both the research work finds that inter-procedural control flow graph based testing method target more fault types than rest of the testing methods. Current dissertation finds this result experimenting in a large scale and can demand more reliability. As a future work, the authors plan to design a testing tool for the testing approach of aspect-oriented software.

**Table 7:** Comparison between the related papers and current dissertation

| | Naqvi et al. (2005) | Parizi & Ghani(2007) | Kumer et al. (2009) | Current dissertation |
|---|---|---|---|---|
| Problem definition | Similar (To provide an analytical overview over current testing methods) | Similar (To provide an analytical overview over current testing methods) | Similar (To provide an analytical overview over current testing methods) | To provide an analytical overview over current testing methods. |
| Testing Methods-used in survey | 3: 1) Data flow based,2) State based,3) Aspect flow graph based. | 5: 1) Data flow based,2) State based,3) Aspect flow graph based, 4) Aspectual behavior based, 5) Model based. | 4: 1) Data flow based,2) State based,3) Aspect flow graph based, 4) Inter-procedural Aspect control flow based. | 18: There are 18 testing methods are surveyed in the current dissertation |
| Coverage – fault types | 6: According to Alexander et al. (2004) fault model | 6: According to Alexander et al. (2004) fault model | 9: Alexander et al. (2004) specify six types of faults and Cecato et al. (2005) add three types. | 10: Alexander et al. (2004) specify six types of faults. Cecato et al. (2005) add three types and one more fault type is identified in recent papers. |
| Method | Survey | Survey | Survey | Survey and there are some experiments in synthesis part. |
| Result | 1) Fault type 2, 3 and 5 are not targeted by any testing method.<br><br>2) The state based testing method target more fault types than other two testing methods. | 1) Fault type 2, 3 and 5 are not targeted by any testing method.<br><br>2) The state based testing method target more fault types than rest of the testing methods. | 1) Fault type 2, 3, 8 and 9 are not targeted by any testing method.<br><br>2) The inter-procedural control flow graph based testing method target more fault types than rest of the testing methods. | 1) Fault type 3 and 9 are not targeted by any testing method.<br><br>2) The inter-procedural control flow graph based testing method target more fault types than rest of the testing methods. |
| Context of synthesis (of analysis) | Only checked method and fault type. | Only checked method and fault type. | Only checked method and fault type. | Not only checked the method and fault type but also checked the relation with traditional techniques and goals of the testing methods. |

# 9 Conclusion

This section contains contributions of this dissertation and suggestion for the future work. A survey over current methods for aspect-oriented software is conducted which also provide a presentation of the state-of-the-art. In this dissertation an analysis about the testing methods of aspect-oriented software is presented. By comparing and contrasting the methods we can draw the following conclusions.

## 9.1 Research Contributions

Current dissertation opens a research area that still to invent testing method to test two kinds of fault types: 1) failure to establish expected post-conditions and 2) incorrect changes in the polymorphic calls. None of the proposed testing approach targets this two fault types. Therefore, one of the important contributions of the current dissertation is to identify this research area where there is a need for more research.

Another important contribution of the current dissertation is that it provides an overview of the techniques. This overview can be useful in case of deciding or finding a specific technique to use. There are lots of software testers using current testing techniques for testing software. The dissertation provides an idea about the relations between traditional techniques and the testing methods of aspect-oriented software. The investigation can help the researchers also by providing idea to investigate a combination of testing approaches of aspect-oriented software that would be cost effective. If a tester intends to test any specific portion of the aspect-oriented software, then he/she can choose the proper testing method from the current dissertation. A tester can choose a testing technique instead of trying all the techniques that are based on similar approaches. This dissertation can help the testers to know about the goal and the limitations of the current testing approaches of the aspect oriented software.

## 9.2 Future Work

There are a lot of on-going researches on testing methods for aspect-oriented software. Therefore, it is interesting to do a follow-up of the current study to see what has happen five years later. The first future work of this dissertation is to incorporate new testing approaches that are in process of research. Future enrichment of the survey can also be helpful to develop the analysis about the testing approaches for aspect oriented software.

The current testing methods are unable to target two fault type of the fault model. The fault types are: 1) failure to establish expected post-conditions and 2) incorrect changes in the polymorphic calls. The second future work is to conduct a research for testing method that will target these two fault types.

As a single proposed testing method has not complete testing feature for testing aspect-oriented software, it is required to develop a testing strategy that will combine several testing methods to get the best effort while testing aspect-oriented software. The third future work is to study combinations of methods from a complementary perspective, i.e., which combinations would be most cost-effective.

# Acknowledgements

First, I want to thank my thesis supervisor Birgitta Lindström for her support, instruction, guidance, advice, encouragement and also for her great patience. Without her help it was not possible to follow the proper scientific methodology for the current dissertation. Finally, I want to thank the thesis examiner Jonas Mellin for his important feedback in the initial stage and also in the final stage.

# References


Alexander, Roger T., Bieman, J.M. & Andrews, A.A. (2004)"Towards the Systematic Testing of Aspect- Oriented Programs", http://www.cs.colostate.edu/~rta/publications/CS-04-105.pdf, 2004.

Alexander, Roger T. & Offutt, Jeff (1999) "Analysis techniques for testing polymorphic relationships" In *Thirtieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '99)*, pages 104–114, Santa Barbara,CA, August 1999.

Ammann, P. & Offutt, J (2008), "Introduction to Software Testing" Cambridge University Press. ISBN: 978-0-521-88038-1.

Anbalagan, Prasanth & Xie, Tao (2006)"Efficient Mutant Generation for Mutation Testing op Pointcuts in Aspect-Oriented Programs.", Second Workshop on Mutation Analysis.

Anbalagan, Prasanth & Xie, Tao (2008) "Automated Generation of Pointcut Mutants for Testing Pointcuts in AspectJ Programs" 19th International Symposium on Software Reliability Engineering, 2008. ISSRE 2008.

Balcer, M., Hasling, W., & Ostrand. T. "Automatic generation of test scripts from formal test specifications" In *Third Symposium on Software Testing, Analysis, and Verification*, pages 210–218, KeyWest Florida, December 1989. ACM SIGSOFT 89.

Beizer, B. (1990),"Software Testing Techniques", Van Nostrand Reinhold. ISBN: 0-442-20672-0.

Bernardi, Mario Luca & Lucca, Giuseppe Antonio Di (2007) "An Interprocedural Aspect Control Flow Graph to Support the Maintenance of Aspect Oriented Systems", IEEE International Conference on Software Maintenance, 2007. ICSM 2007.

Bernardi, Mario Luca & Lucca, Giuseppe Antonio Di (2007) "Testing Aspect Oriented Programs: an Approach Based on the Coverage of Interactions among Advices and Methods." Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference.

Binder, Robert V. (1994) "Design for testability in object-oriented systems" Communications of the ACM, 37(9):87–101, September 1994.

Buy, Ugo, Orso, Alessandro & Pezze, Mauro (2000) "Automated testing of classes" In *2000 International Symposium on Software Testing, and Analysis (ISSTA '00)*pages 39–48, Portland, OR, August 2000. IEEE Computer Society Press.

Ceccato, Mariano, Tonella, Paolo & Ricca, Filippo (2005)"Is AOP code easier or harder to test than OOP code?" proceedings of working on testing Aspect oriented programs 4th international conference on AOSD Chicagtog llinois.

Collofello, Jim & Vehathiri, Kalpana (2005), " An Environment for Training Computer ScienceStudents on Software Testing", Frontiers in Education, 2005. FIE '05. Proceedings 35th Annual Conference.

Dahl, O.-J., Dijkstra, E. W. & Hoare, C.A.R. (1972), "Structured Programming" London: Academic Press. ISBN: 0-12-200550-3.

Delamare, Romain, Baudry, Benoit & Traon, Yves Le (2009), "AjMutator: A Tool For The Mutation Analysis Of AspectJ Pointcut descriptors" International Conference on Software Testing, Verification and Validation Workshops, 2009. ICSTW '09.

Ferrari, Fabiano Cutigi, Maldonado, Jose Carlos & Rashid, Awais (2008) "Mutation testing for aspect-oriented programs" 2008 International Conference on Software Testing, Verification, and Validation.

Freedman, Roy S. (1991) "Testability of software components" *IEEE Transactions onSoftware Engineering*, 17(6):553–564, June 1991.

Hamlet, R. (1994) "Random testing," Encyclopedia of software Engineering,

Harrold, Mary Jean & Soffa, Mary Lou (1991) "Selecting and using data for integrationtesting" *IEEE Software*, 8(2):58–65, March 1991.

Jacobsen, I. & Ng, P-W. (2004). "Aspect-oriented Software Development with Use Cases" Boston: Addison-Wesley.

Kaur, Charnpreet & Garg, Sushil (2012) "Testing aspect oriented software using uml activity diagram." International Journal of Engineering Research & Technology (IJERT).

Kummar, Mohit, Sharma, Akashdeep & Grag, Susil (2009), "A Study of Aspect Oriented Testing Techniques.", IEEE Symposium on Industrial Electronics and Applications (ISIEA 2009), October 4-6, 2009, Kuala Lumpur, Malaysia.

Legard, H. & Marcotty, M. (1975) "Ageneralogy of control structures." *Communications of the ACM*, 18:629–639, November 1975

Li, Xiaofei & Xie, Xusheng (2009), "Research of Software Testing Based on AOP", 2009 Third International Symposium on Intelligent Information Technology Application.

Lindström, Birgitta (2000) "Methods for Increasing Software Testability" Student Thesis, Institutionen för datavetenskap, University of Skövde.

Lopes, Cristina Videira & Ngo, Trung Chi (2005), "Unit-Testing Aspectual Behavior" In preceedings of the 1st workshop on Testing Aspect-Oriented Programming.

Moreira, Rodrigo M. L. M., Paiva, Ana C. R. & Aguiar, Ademar (2010)"Testing Aspect-Oriented Programs" Information Systems and Technologies (CISTI), 2010 5th Iberian Conference.

Naqvi, Syed Asad Ali, Ali, S. & Khan, M. U. (2005)"An Evaluation of Aspect Oriented Testing Techniques", In Proceedings of International Conference on Emerging Technologies, Mohammad Ali Jinnah University, Islamabad, Pakistan, IEEE.

Osterweil, L. J. & Fosdick, L. D. (1974) "Data flow analysis as an aid in documentation, assertion generation, validation, and error detection" Technical report CU-CS-055-74, Department of Computer Science, University of Colorado, Boulder, CO.

Parizi, Reza Meimandi & Ghani, Abdul Azim (2007), "A Survey on Aspect-Oriented Testing Approaches", Fifth International Conference on Computational Science and Applications.

Parizi, Reza Meimandi, Ghani, Abdul Azim, Abdullah, Rusli & Atan, Rodziah (2009) "On the applicability of Random Testing for aspect-oriented programs" International Journal of Software Engineering and its Applications. Vol. 3, No. 4, October, 2009.

Radnoci, Ramon (2009) "Methods for Testing Concurrent Software" Student Thesis, Institutionen för datavetenskap, University of Skövde.

Schütz, W. (1994), 'Fundamental issues in testing distributed real-time sys-tems',Real-Time Systems7(2), 129–157.

Sommerville, Ian (2010), "chapter-21: Aspect-oriented software engineering (Software Engineering (9th Edition)", ISBN-13: 978-0137035151)

Ubayashi, Naoyasu & Tamai, Tetsu (2002) "Aspect-Oriented Programming with Model Checking", AOSD '02 Proceedings of the 1st international conference on Aspect-oriented software development.

Weyuker, Elaine J. (2009), "Can We Measure Software Testing Effectiveness?" Software Metrics Symposium, 1993. Proceedings, First International.

Xu, Dianxiang, El-Ariss, Omar, Xu, Weifeng & Wang, Linzhang (2010) "Testing aspect-oriented programs with finite state machines" Published online 13 September 2010 in Wiley Online Library (wileyonlinelibrary.com).

Xu, Dianxiang & Xu, Weifeng (2006) "State-based incremental testing of aspect-oriented programs" Proceedings of the 5th international conference on Aspect-oriented software development

Xu, Dianxiang, Xu, Weifeng & Nygard, Kendall (2004) "A State-Based Approach to Testing Aspect-Oriented Programs", In Proc. 17th International Conference on Software Engineering and Knowledge Engineering.

Xu, Dianxiang, Xu, Weifeng & Wong, W. Eric (2008) "Testing aspect-oriented programs with uml design models." International Journal of Software Engineering and knowledge Engineering.

Xu, Weifeng & Xu, Dianxiang (2005)"A Model-Based Approach to Test Generation for Aspect-Oriented programs" AOSD Workshop on Testing AOP.

Xu, Weifeng, Xu, Dianxiang, Goel, Vivek & Nygard, Ken (2004)“Aspect Flow Graph For Testing Aspect-Oriented Programs” http://www.cs.nodak.edu/-wxu/research/436-11 ljd.pdf

Zhang, Sai & Zhao, Jianjun (2007)“On Identifying Bug Patterns in Aspect-Oriented Programs”, Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International.

Zhao, Chuan & Alexander, Roger T. (2007) “Testing Aspect-Oriented Programs as Object-Oriented Programs” Proceedings of the 3rd workshop on Testing aspect-oriented programs

Zhao, J. (2003) “Data-Flow-Based Unit Testing of Aspect-Oriented Programs”, In Proc.27th Annual IEEE International Computer Software and Applications Conference (COMPSAC'2003), Dallas, Texas,December 2003, pp. 188-197.

Zhou, Yuewei, Ziv, Hadar & Richardson, Debra (2004), “Towards A Practical Approach to Test Aspect-Oriented Software”. *Workshops on Testing* of Component-Based Systems (TECOS ... GI *2004* P-58, 1-16 (*2004*)