Aspect-Oriented Program Testing: An Annotated Bibliography

[1]Abdul Azim Abdul Ghani and [2]Reza Meimandi Parizi
Department of Information System
Faculty of Computer Science and Information Technology
Universiti Putra Malaysia
43400 Serdang, Selangor, Malaysia
{[1]azim@fsktm.upm.edu.my, [2]r.m.parizi@gmail.com}

***Abstract*—Research in aspect-oriented software testing has resulted in many approaches as reported in literature. A few papers have devoted to literature survey in this field of research. However, the survey only focuses on certain selected topic and particular approaches rather than providing a comprehensive set of references that cover most of the work related to aspect-oriented software testing as a whole. In this case, there is no work yet reported in the literature to tackle this shortage. Therefore, in this paper a collection of 81 references drawn from journals, conference and workshop proceedings, thesis, and technical reports on the subject of testing aspect-oriented software is presented. Each reference is accompanied by a summary of important finding. The aim when selecting the references was to cover as many related articles starting from the first work on the subject in 2002 until the year 2011. For this reason, the bibliography is intended to help the researcher or practitioner, who is relatively new, in gathering information on the subject. The bibliography is organized according to the following sections: general introduction; background on the subject; issues in testing aspect-oriented software; fault models and types; testing coverage criteria; aspect-oriented testing techniques; and automated support for testing aspect-oriented software.**

***Index Terms*—Software testing, Testing survey, Aspect-oriented programming (AOP), AOP testing**

I. INTRODUCTION

Building large and complex software applications is a challenging task for software engineers. Besides adhering to complex functionalities, software engineers need to build software applications that conform to non-functionality requirements such as quality factors of software. Coping with complexity and achieving quality software is a major issue in building software applications. One of the fundamental principles in software engineering for handling the issue is the separation of concerns principle, for a better modularity of code. Typically, a *concern* is a feature or required property that is specified in a requirement model for the software. The principle states that any complex problem can be more easily dealt with if it is subdivided into different kinds of concerns that can be solved independently in different modules.

Over the years, software engineering has experienced an evolution of various types of development paradigms and programming languages that have offered useful mechanisms to handle modularity. The introduction of the procedural programming paradigm has provided software engineers with abstraction mechanisms for them to structure the software into separate but cooperating modules. The emergence of object-oriented programming (OOP) paradigm enhances further the ability of software engineers to design and to program with modularity in mind using object-oriented features such as class, inheritance, delegation, encapsulation and polymorphism. Nevertheless, practically, these programming paradigms are inherently unable to modularize all concerns of interest for complex software systems since some concerns crosscut a broader set of modules, known as *crosscutting concerns*, which could not be easily specified in single modules. Aspect-oriented programming paradigm is the next that emerges to enhance software development in better handling of separation of concerns.

Aspect-oriented programming (AOP) [82] is a technology that was first introduced in the middle of 1990s at the Xerox Palo Alto Research Center. The purpose of AOP is to improve separation of concerns by providing explicit concepts to modularize the crosscutting concerns. AOP uses some improved abstractions/constructs to represent concerns that crosscut the program modules. Some examples of typical crosscutting concerns are security, synchronization policies, and logging, that span the entire systems. Ideally, each crosscutting concern can be designed and implemented independently. AOP separates crosscutting concern from the rest of the code (*core concern*) into named modules called *aspects*. It is claimed that by doing this, the cohesion and reusability of the classes that implement the core concerns will be increased, thus will increase the overall quality of software.

However, AOP alone to increase the quality of software does not guarantee developers and programmers from introducing mistakes. As consequence aspect-oriented programs produced will not be error free. Its new concepts, e.g. constructs and properties, bring new challenges and aspect-related faults not present when testing other types of programs. In other words, testing

doi:10.4304/jsw.8.6.1281-1300

aspect-oriented programs could not be directly performed using the current testing techniques used on other programming paradigms, e.g. object-oriented. Thus testing remains as an important activity in aspect-oriented software development.

Over the years, testing aspect-oriented programs has gained considerable interest from researchers. Over 80 research literature items on this topic have been identified in which the research conducted are generally related to either: (i) new testing approaches that are being leveraged or extended based on traditional techniques; or (ii) new testing criteria with respect to fault types due to aspect-oriented concepts. A few of the literature [13][38][48][64][69] are dedicated to reviews and surveys on the topic, however none of these studies provides a comprehensive set of references that cover most of the work related to aspect-oriented software testing as a whole. In this regard, this bibliography has grown out to make an inventory of testing aspect-oriented programs and provides references to researchers working in this topic. The papers listed are annotated with summaries, which in turn are cross-referenced to related papers. References 1 to 81 are directly related to testing aspect-oriented programs, while the rest of the references are used as the background for the topic.

The rest of this paper is organized as follows: Section II presents the background on the topic consisting of software testing and aspect-oriented programming concepts; Section III discusses issues on testing aspect-oriented programs; Section IV presents the fault models and fault types for aspect-oriented programs; Section V discusses coverage criteria for testing aspect-oriented programs; Section VI presents the techniques that have been proposed; Section VII presents automated tools for testing aspect-oriented program; Section VIII presents empirical studies conducted in testing aspect-oriented programs, and Section IX presents concluding remarks.

II. Background

This section provides general information consisting of concepts and terminology on aspect-oriented programming, AspectJ language, and terminology and techniques in software testing.

A. Aspect-oriented Programming Concepts

This sub-section introduces the concepts and idea behind aspect-oriented programming. It briefly describes basic concepts that are introduced in this programming paradigm [87]. The detailed description on the concept could also be obtained from the AspectJ Team webpage located at http://www.eclipse.org/aspectj/doc/released/progguide/index.html. The webpage also contains the programming guide for AspectJ language, which is the most commonly used AOP language in practice.

AOP is a programming paradigm that allows for separation of crosscutting concerns. AOP supports the implementation of crosscutting concern into named modules called *aspects* that each of them encapsulates a crosscutting concern. An aspect is similar to class in object-oriented programming (OOP). Besides having the properties of a class in OOP, an aspect encapsulates the behavior, and state of a crosscutting concern. In AOP languages, aspects can only be invoked at well-defined points in the execution of a program. These points are called *join points*. Examples of joint points are calling or execution of methods, access to an attribute, and initialization of an object. Join points can be determined in a *pointcut* or *pointcut designator*. A pointcut describes a set of join points where an *advice* needs to be invoked.

An advice is a method-like construct that contains behavior to execute at a matched joint point. For example, this might be the security code to do authentication and access control. The advice is woven into the join points when a pattern of a pointcut is matched. In other words, an advice is used to express the crosscutting actions that must take place within the method body at the matched join point. There are three kinds of advices: before advice, after advice, and around advice.

Since there are aspect and non-aspect code (i.e. base code) in a program, the aspect code must be run properly with the non-aspect code. This can be realized through *aspect weaving*. Aspect weaving is the process by which behavior on aspects are merged to the core concern code to yield a working system. Several mechanisms for weaving have been defined depending on AOP languages. These include statically compiling the advice together with base code, dynamically inserting aspects when loading code, and modifying the system interpreter to execute aspects. For example, in AspectJ aspect weaving composes the code of the base code and the aspects to ensure that applicable advice runs at the appropriate join points. Fig. 1 shows the generic AOP process.

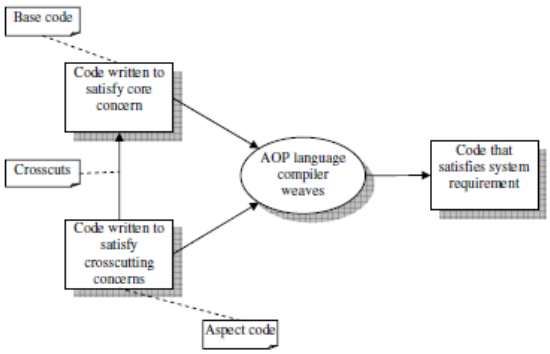

Figure 1. Generic AOP process.

B. AspectJ Language

AspectJ, an extension of Java language, is developed to support aspect-oriented programming. It is the most popular AOP language to date and most of the aspect-oriented testing papers base their work on AspectJ language. AspectJ realizes crosscutting constructs in AOP by offering common crosscutting constructs, dynamic crosscutting construct, and static crosscutting constructs [87]. These constructs are the basis for forming the building block of AspectJ programs. Common crosscutting constructs consist of join point, pointcut, and

```
1. public class Point {
2.   protected int x, y;
3.  public Point(int _x, int _y) {
4.       x = _x;
5.       y = _y;
    }
6.  public int getX() {
7.       return x;
    }
8.  public int getY() {
9.       return y;
    }
10. public void setX(int _x) {
11.      x = _x;
    }
12. public void setY(int _y) {
13.      y = _y;
    }
14. public void printPosition() {
15.      System.out.println("Point at
         (" + x + "," + y + ")");
    }
16.  public static void main(String[]
     args) {
17.      Point p = new Point(1, 1);
18.      p.setX(2);
19.      p.setY(2);
    }
   }
20. class Shadow {
21.  public static final int offset = 10;
22.  public int x, y;

23.  Shadow(int x, int y) {
24.      this.x = x;
25.      this.y = y;
26.      public void printPosition() {
         System.out.println("Shadow at
         (" + x + "," + y + ")");
     }
    }
```

(a) base code

```
27. aspect PointShadowProtocol {
28.   private int shadowCount = 0;
29.   public static int getShadowCount(){
30.    return PointShadowProtocol.
                aspectOf().shadowCount;
      }
31.   private Shadow Point.shadow;
32.   public static void associate(Point p,
         Shadow s){
33.        p.shadow = s;
      }
34.   public static Shadow getShadow(Point p) {
       return p.shadow;
      }
35.   pointcut setting(int x, int y, Point p):
        args(x,y) && call(Point.new(int,int));
36.   pointcut settingX(Point p):
       target(p) && call(void Point.setX(int));
37.   pointcut settingY(Point p):
       target(p) && call(void Point.setY(int));

38.   after(int x, int y, Point p) returning :
          setting(x, y, p) {
39.      Shadow s = new Shadow(x,y);
40.      associate(p,s);
41.      shadowCount++;
      }
42.   after(Point p): settingX(p) {
43.      Shadow s = new getShadow(p);
44.      s.x = p.getX() + Shadow.offset;
45.      p.printPosition();
46.      s.printPosition();
      }
47.   after(Point p): settingY(p) {
48.      Shadow s = new getShadow(p);
49.      s.y = p.getY() + Shadow.offset;
50.      p.printPosition();
51.      s.printPosition();
      }
    }
```

(b) aspect code

Figure 2. A sample AspectJ program.

advice. Dynamic crosscutting construct is achieved through the support of advice that modifies the behavior of a program. Whereas, static crosscutting constructs are in the form of intertype declarations and weave-time declarations, modify the static structure of a program. In terms of aspect-oriented program testing, the interests are in testing the behavior of dynamic crosscutting construct.

The related keywords in AspectJ are `aspect`, `before`, `after`, and `around` advice, and `pointcut`. An `aspect` can contain the code specifying pointcuts, different kinds of advice, and intertype declarations (an aspect declares another types; can be an interface, a class or an aspect). `pointcut` is used to define a named pointcut for join point in a program. The keywords `before`, `after`, and `around` advice are method-like constructs consisting of code used to specify crosscutting behavior at join points. A `before` advice executes its body before executing the body of the matched join point. An `after` advice executes its body after executing the body of the matched join point. An `around` advice body surrounds the match join point. It may change the execution of the matched join point body, or may even replace the matched join point body. Example program written in AspectJ, taken from [3], in Fig. 2 shows the related constructs.

The program in the figure is divided into two parts: (a) the base code and (b) the aspect code. The base code contains the classes `Point` and `Shadow` at line 1 and 20 respectively, whereas, the aspect code contains the aspect `PointShadowProtocol` at line 27. In the figure, the aspect `PointShadowProtocol` defines three pointcuts that are `setting` at line 35, `settingX` at line 36, and `settingY` at line 37. The aspect `PointShadowProtocol` also specifies three kinds of after advice which are attached to their corresponding pointcuts `setting`, `settingX`, and `settingY` as shown at line 38, 42, and 47 respectively.

C. Software Testing

Software testing is an important activity in any software development process. It is an 'umbrella' to all phases in the process. However, there is no single agreeable definition for software testing. Software testing is claimed as a process of executing a program with the intention to find faults [83]. It is also defined to involve any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets with its required results [85]. Another definition given is as evaluating software by observing its execution [86]. Nevertheless, the essence of software testing is to execute the program with a particular set of input and observing the actual output produced then comparing the output produced with the expected output. In other words, it is to determine the quality of a software system by analyzing the results of running it. This particular set of program input along with the corresponding expected output is called a test case. The test cases are generated by using testing techniques.

There are many testing techniques and methods used with different purposes and thus, they are classified differently [88]. Readers may refer to [83][84][85][86] for detail information of the techniques and methods. However, for the benefit of readers who are not familiar with the techniques, a brief account of the techniques is given.

Functional testing technique derives its test cases by analyzing the program's input and output from the program specification, without considering the implementation details of the program. With the advanced in object orientation and models in software engineering, there has been a growth in model-based testing. *Model-based testing* is considered as a type of functional testing technique. In model-based testing, design models used for designing are the basis for performing testing. These models can be used to represent the desired behavior of the System Under Test (SUT). Examples of such models are finite state machine, statecharts, and decision tables. *State-based testing* is one of the model-based testing techniques in which test cases are derived from a state model that describes systems requirements and functionality.

Structural testing technique, on the other hand derives its test cases from the knowledge of program's implementation, e.g. control flow path or use of specific data items. Techniques come under this classification are *control flow testing technique* and *data flow testing technique*. Control flow testing technique requires knowledge of control flow structure from source code. The control flow structure provides paths in a program to be selected for testing purposes. Data flow testing technique requires knowledge from source code to select paths in a program according to sequences of events related to data state. However, structural information can also be gathered from design or specification artifact. Thus, it is convenient to also classify testing technique as *code-based technique* if the test data are generated with the knowledge from source-code.

Another classification is *mutation testing technique*. This technique focuses on modeling faults by means of mutation operators for specific languages. A series of mutants is produced when each mutation operator is applied to a program. Test cases are generated to examine the mutated versions of the program. *Fault-based testing* is another classification to show that the program is not incorrect. If a program has limited potential for being incorrect, then test data demonstrate correctness when they show the potential is not realized. The means of specifying incorrectness taken here is to define potential faults for program constructs which is what is done in mutation testing. Other technique that is used in software testing is *random testing technique*. The technique produces randomly generated test cases.

Besides the techniques mentioned previously, software testing is usually performed at three levels:

(1) *Unit testing*, the smallest units produced by the implementation are tested in isolation. This testing level aims to find fault in the logic and implementation.
(2) *Integration testing*, the interactions between among the units are tested to find faults in the logic and the interfaces.
(3) *System testing*, the assembly or integration of all sub-systems of a system is tested to verify that the system is adequately assembled in producing the expected functions and performance. This level of testing looks at design and specification problems.
(4) *Acceptance testing*, the completed system is user-tested to verify the software does what the users want.
(5) *Regression testing*, the updated version of system is tested after changes are made to the system to ensure it still possess the functionality it had before the changes.

Theoretically to exhaustively test a program is not possible since the number of potential inputs for most programs is effectively infinite. Thus, *coverage criteria* are used to decide which inputs are to be included in a test. A coverage criterion is a rule or collection of rules that impose test requirements on a test set. Types of coverage criteria are determined by the techniques and models used in testing the software.

One goal of software testing is to automate as much as possible in order to reduce testing cost, minimize human errors, and make regression test easier. Testing automation [73] consists of a) automated test input generation and selection, b) automated test oracle, and c) automated test execution. In automated test input generation and selection, tools are used to generate test data as specified by testing techniques in which coverage criteria are the determinants for producing the test data. Automated test oracle will check the correctness of the tests. The runtime behavior of the program under test is automatically checked with respect to the generated test inputs. In automated test execution, a framework allows test to be executed. The framework executes the tests and collects the test results by means of oracles.

III. Issues in Testing Aspect-Oriented Programs

Aspect-oriented programming paradigm has brought along with it new concepts and properties that extend the capabilities of other programming paradigms. In AOP, separate aspects are defined to contain crosscutting actions, that later are woven into classes that represent the core concerns of the system. Because of that, testing aspect-oriented software could not be directly performed using current testing techniques used on other programming paradigms. Such concepts and properties pose new challenges and issues regarding testing. The followings are the issues [2][48][69] [81] faced in testing aspect-oriented programs:

- **Aspects do not have independent identity**. Aspects depend on the context of their use in a program with respect to the base classes. Thus, an aspect could not be separately tested as a unit. The aspect needs to be woven together with its base classes to generate executable code before testing it.

- **Aspects can be tightly coupled to their woven context.** Aspects are often tightly coupled with their woven classes. Thus, any change to these classes will likely impact the aspects.

- **Control and data dependencies are not readily apparent from the source code of aspects or classes**. The nature of weaving process results in difficulty in comprehension of control and data dependencies by developers. Thus, relating failures to corresponding faults becomes difficult.

- **Interaction between classes and aspects results in emergent behavior**. The root cause of a fault may lie in a class, or an aspect, or it may be as a side effect of a particular weave order of multiple aspects. Thus, resulting is potential faults that are difficult to diagnose.

- **Behavioral changes due to foreign aspects.** The use of foreign aspects in a software system can introduce unexpected and undesired behavior. Thus, they can affect program correctness, comprehension, and maintenance.

- **Interference of aspects can result conflicting behavior.** The introduction of different types of changes by aspects into a software system produces different types of interferences that the aspects can cause.

- **Problems in pointcut descriptors (PCDs) if they are wrongly formulated.** Faults will be injected due to wrong formulation of PCDs by developers. This introduces additional behavior or fails to be applied to related join points.

Besides the above issues, other issues that can influence aspect-oriented program testing are undisclosed type of errors or bug patterns, and recurring or symptomatic issues [48][69].

The above issues have resulted in many attempts by researchers in proposing new or extended techniques for testing aspect-oriented programs. The traditional testing techniques, while applicable to certain extends in testing core classes, are not directly applicable to test aspects. In the following sections, work on testing techniques on aspect-oriented programs is described.

IV. Fault Models and Fault Types for Aspect-Oriented Programs

A *fault model* determines the types of faults that components of a system under test most likely to have. The AOP fault model helps in understanding how faults and failures occur in aspect-oriented programs. Almost all approaches described in the next section have their proposed methods work on certain fault models or types. The first work on fault model for aspect-oriented programs is the contribution of [2]. The fault model proposed is based on the nature of faults and unique properties of AOP which are related to structural and behavioral properties. However, the challenge is, when a failure occurs, diagnosing the failure and detecting source of fault would not be trivial. Source of fault is a location in a program that a fault may have occurred. The potential sources of faults in an AOP program are listed below [2]:

- A fault resides in a portion of the core concern not affected by an aspect.
- A fault resides in code that is related to an aspect, isolated from the woven process.
- A fault is related to an emergent property created from interactions of one aspect with the primary abstraction
- A fault is related to an emergent property created from interactions of multiple aspects with the primary abstraction.

It is important to note that, some of the issues listed in Section III (i.e. 1-3) are resulting from the nature of AO paradigm and its associated properties (i.e. obliviousness). This can make it to say that not all the above fault sources can be easily map to the issues list in previous section, for instance source 3 and 4 can be mapped well to the issues list whereas other cannot be.

The fault model proposed in [2] has listed different fault types specifically for AO programs. The fault types are faults classified based on the characteristics of AOP. They are:

- Incorrect strength in pointcut patterns
- Incorrect aspect precedence
- Failure to establish expected postconditions
- Failure to preserve state invariants
- Incorrect focus of control flow
- Incorrect changes in control dependencies

Additional fault types proposed are pointcut descriptor related faults [26], bug patterns [40], fault related to foreign aspects [80], faults related to AspectJ pointcut [29][30] and advice [31], faults related to intertype declaration, pointcut, and advice [18], faults related to interactions among methods and advice by means of the pointcut expressions defining the join points [37], fault types related to interactions derived with the help of interaction model and dependency model [34], fault types as the basis for the construction of mutation operators for aspect-oriented program [46] [68], faults that occur during aspect composition from sequences diagrams [53], and faults related to state-based aspect design [71].

The work of [46] has produced comprehensive fault types. The types are (1) pointcut expressions; (2) intertype declaration and other declarations; (3) advice definition and implementation; and (4) base program. Empirical analysis based on recurring observed scenario on the fault types has resulted in a more refine categories [70]. Other types suggested are based on mistakes made by programmers during refactoring of crosscutting concerns [78] and a fault model associated with risk assessment [55]. Other additional fault types can be found in [17].

Most work on fault type focus mainly on the intuition that faults may be caused by pointcuts. However, an empirical study reveals contradiction, in which other mechanisms such as intertype declaration and advice also contribute to having faults in aspect-oriented programs [66]. Besides the empirical studies specific in analyzing the faults occurrence in aspect-oriented programs, fault types proposed by various research papers are also used in empirical studies as identifying factors for effectiveness of certain testing approaches with respect to testing criteria. Examples of such study are [74][71][75].

V. Coverage Criteria for Testing Aspect-oriented Programs

It is nearly impossible to enumerate all inputs for testing the programs since the number of potential inputs for most of the programs is so large that it could be infinite. Because of that, coverage criteria are used to decide which inputs are to be included in a test. The effective use of coverage criteria tends to help testers in uncovering faults in a program. Those criteria are used to measure the coverage of a test suite in terms of percentage. Practically, coverage criteria are indicators for when to stop testing.

In testing techniques and methods used to test software written in traditional or object-oriented paradigms, coverage criteria have been so helpful in identifying test cases. Usually, coverage criteria are related to underlying models used or approaches in testing a program. However not all coverage criteria are directly useful for testing aspect-oriented programs as the nature of aspect-oriented programs needs different kind of models or extension of current models for testing. As far as the literature is concerned, the coverage criteria for testing aspect-oriented programs fall under code-based criteria, model-based criteria, or fault-based criteria (mutation testing). The criteria are described along with their testing technique or approach. The next section describes the approaches.

In code-based coverage criteria, aspect-oriented program source code or Java bytecode is used as the basis for defining coverage criteria, or some forms of graph model are the basis for defining the coverage criteria. Coverage criteria defined based on source code are (1) a set of aspect coverage criteria which include statement coverage, insertion coverage (also known as joinpoint coverage [44] or all-crosscutting-node criteria [32]), context coverage, and def-use coverage [5][11]; (2) aspectual branch coverage, interaction coverage, dataflow coverage, and data coverage [22].

Other perspective of defining coverage criteria is based on flow graphs. The source code will be mapped into a graph-based model to describe the control flow model or data flow model. The most common flow graph to represent control flow model is control flow graph (CFG) in which a node represents a statement and an edge represents a control flow from one node to another. The data flow model is used to model the flows of the data values in source code. However, flow graphs are not only constructed from source code, they could also be constructed from other modeling artifact such as design artifact. Then, coverage criteria are defined on those flow graph models. In the case of aspect-oriented programs, new flow graph models are proposed to handle the representation of aspects and their integration to base programs. Thus, new coverage criteria are defined based on the proposed graphs.

One such flow graph that is used to define coverage criteria is aspect-oriented def-use graph model (*AODU*) [32]. The model is an extension of the original def-use model for object-oriented program in which the advice interactions that occur in aspect-oriented programs are represented by a set of nodes affected by pieces of advice called crosscutting nodes. The model is meant for unit testing aspect-oriented Java programs. Besides the traditional def-use criteria, the aspect-oriented testing coverage criteria defined on the model are all-crosscutting-nodes, all-crosscutting-edges, and all-crosscutting-uses. *AODU* graph model has been used as the basis for other types of models. One such model is *PWDU* (PairWise Def-Use) graph [43] [63] that is used to represent the structure of a pair of units that interacts with each other in integration testing approach for object-oriented and aspect-oriented programs. The interacting units can be: i) when a method calls another, ii) when a method is affected by an advice, iii) when an advice calls a method, and iv) when an advice is affected by another advice. The coverage criteria defined on the model are all-pairwise-integrated-node, all-pairwise-integrated-edges, and all-pairwise-integrated-uses. Another model known as *PCCFG* (Pointcut-based Control Flow Graph) is used to define coverage criteria for testing each advice-pointcut pair [51]. However, this model covers only control flow pointcut-based criteria. A model called Pointcut-based Def-Use Graph (*PCDU*) [74] is used to

model both the flow of control and data at join points. An additional coverage criterion related to data flow between advice and pointcut is proposed. Another set of testing coverage criteria for AO software is based on Inter-procedural Aspect Control Flow Graph (IACFG) [37] [50]. These coverage criteria cover the interactions among aspects and classes during integration testing. A set of dataflow coverage criteria for AO is also defined using a framed Inter-procedural Control Flow Graph (ICFG) [65]. It covers interactions that are based on state variables.

Coverage criteria are also defined based on flow graphs constructed from aspect-oriented UML models. Aspect-oriented model may consist of class diagrams, aspect diagrams, statecharts, collaboration diagrams, and sequence diagrams. Coverage criteria are proposed by taking into consideration the integration of aspects in a collaboration diagram [19]. A control flow graph is used to modularize the control of the methods involved in the collaboration. Other coverage criteria proposed are based on dynamic behavior modeled in an extended statechart [15]. Those coverage criteria are classified under transition coverage criterion sequence coverage criterion, advice execution coverage criterion, and multi-aspect integration coverage criterion. Besides work that proposed new set of coverage criteria, existing coverage criteria for UML diagrams (use case coverage, transition coverage, state coverage, polymorphic coverage) are used in test generation of aspect-oriented programs. The existing coverage criteria are used on aspect flow graph [6] [9], aspect-object flow tree [47], and aspectual use cases [42].

VI. Techniques in Testing Aspect-oriented Programs

Various techniques have been proposed by researchers in testing aspect-oriented programs. There are also a few categories have been put forward in classifying the techniques [56][61][64]. This paper takes into consideration the categories, and focuses on the artifact that the techniques used as the basis for testing aspect-oriented programs. The techniques are classified under code-based testing, model-based testing, fault-based testing focusing on mutation testing, regression testing and other approaches.

Code-based testing technique emphasizes the generation of test data from the knowledge of aspect implementation which involves the base code and aspect code. The knowledge obtained from this implementation is in the form of internal structure that is structurally represented by using control flow graph or data flow graph. Most work focus on testing source code written in AspectJ language as their underlying language. The earliest work are based on control flow and data flow models [1][3]. Derivations of the original control flow and data flow graph models are proposed in testing aspect-oriented programs. They are aspect-oriented def-use graph model (*AODU*) [32], *PWDU* (PairWise Def-Use) graph [43] [63], *PCCFG* (Pointcut-based Control Flow Graph) [51][52], Pointcut-based Def-Use Graph (*PCDU*) [74], Inter-procedural Aspect Control Flow Graph (IACFG) [37] [50] and Inter-procedural Control Flow Graph (ICFG) [65].

Model-based testing technique focuses on deriving test cases partially or fully from aspect-oriented models. The aspect-oriented models consider the behavior of programs when aspects are interacting with classes. One of the techniques is state-based technique in which finite state machine is extended to include the aspects for describing the aspectual behavior [8][21][41]. The technique defines new state model considering aspect-related properties, from which state transition trees and test cases are derived. Aspect-oriented state model is an extended model that is used in MACT (Model-based Aspect/class Checking and Testing) framework [71] [72] in which structure-oriented and property-oriented testing strategies are employed to automatically generate test cases. Another state-based technique [49] uses an aspect object state diagram to model crosscut weaving model, which is transformed into a tree to derive test cases. Other testing techniques under model-based techniques use extended UML diagrams as their basis for deriving test cases. The techniques are based on UML collaboration diagrams [19] [20] [36], UML state diagrams [15] [60], aspect-oriented UML design models (consist of class diagrams, aspect diagrams and sequence diagrams) [9] [47], UML sequence diagrams [53], and aspectual use cases [42].

Mutation testing technique for aspect-oriented programs focuses on modeling faults by means of mutation operators for aspect-oriented languages. A series of mutants is produced when each mutation operator is applied to a program. Test cases are generated to examine the mutated versions of the program. The mutation testing techniques for testing aspect-oriented programs include automated generation of mutant for testing pointcuts [25] [45], testing fault related to pointcut descriptors [26] [57] [58] [75], automated generation of mutants for AspectJ programs [62] [67], and definition of a comprehensive set of mutation operators [5] [11] [46] [79].

As aspect-oriented program is modified, it is required to be regression tested so that the changes do not introduce new faults to the original program. Since the size of test suite typically keeps growing, regression test needs to be properly conducted and its test selection techniques can be employed to reduce cost of testing. There is work that studied that focus on the regression testing of aspect-oriented programs [27] [28] [39]. Basically this work depends on structural information of source code in the form of control flow graph. In [39], AspectJ Inter-module Graph (AJIG) is proposed to regression test AspectJ programs. AJIG consists of CFG and interaction graphs.

Further work in testing techniques for aspect-oriented programs involves random testing and search-based testing. In random testing of aspect-oriented programs, an automated random-based test generation and test execution [56] [61] and a work on adaptive random testing for AOP [76] are proposed. In search-based testing, an optimization approach to automate test data

generation for structural coverage of AOP systems is introduced [54]. Other work is proposing application specific testing aspects identified as test oracles [7]. It uses Aspect-Oriented Test Description Language (AODTL) to build the testing aspects which later is used to generate test oracles. Double-phase testing method is proposed to eliminate meaningless test case.

VII. Automated Support for Testing Aspect-oriented Programs

Two perspectives on automated testing of aspect-oriented programs provide roadmap for classifying automated testing. One perspective is along test-input generation and selection [33], and the other perspective is along three levels of automation [73] as mentioned in Section II.C. In whatever perspectives, automated support for testing is very important in order to reduce testing cost, minimize human errors, and make regression test easier. In addition, as the theory and practice of aspect-oriented programming is becoming more mature, automated support for testing such programs is needed.

Since the first work on aspect-oriented software testing, a very few number of automated approaches have been proposed. Automated approaches that adopt Java test-generation tools for generating test data in testing AspectJ programs are Wrasp [14], Aspectra [22], and Raspect [23]. Wrasp automatically generates both unit and integration tests for AspectJ programs focusing on aspectual behavior and in addition adopts JamlUnit [10] and AJTE [16]. Aspectra uses a similar approach to Wrasp, but focuses on automatic generation of test inputs to test aspectual behavior. Raspect complements Wrasp and Aspectra in which it detects redundant unit test. APTE [24] is an automated framework, that tests pointcuts in AspectJ program, is built on another tool AJTE [16]. AJTE is a tool to unit test without weaving. Another new tool is EAT [54] that uses search-based optimization approach to automate test data generation.

Automated support for mutation testing also has seen a number of proposed tools. Initially a tool that implement mutation analysis on pointcut expression [25][45]is proposed. However, *AjMutator* [57] tool is proposed to include more mutation operators as defined in [46]. This tool works together with AdviceTracer[58] [75], a tool to specify an oracle that expects the presence and absence of an advice at a specific point in the base program. *Proteum/AJ* [67] is another tool that improves the previous tools by adding new functionalities such as mutation operator selection. Another available tool is MuAspectJ [62] that supports a full range of mutation operators on AspectJ. It is an extension of MuJava (this tool is to generate mutant for Java language).

In the perspective of test oracle, an approach that produces JAOUT tool to generate test codes to serve as test oracles is proposed [7]. The approach makes use of Aspect-Oriented Test Description Language (AOTDL) to help build testing aspects that are translated by JAOUT. Automated tools are also produced to support the model-based testing approaches. AJUnit [60], a tool based on JUnit, is used to generate testing sequences covering an aspect-class block of code in for UML Statechart Diagrams. MACT (Model-based Aspect/class Checking and Testing) [71] is another model-based tool, works as a framework, to generate test cases from aspect-oriented state-model.

VIII. Empirical Studies Conducted on Aspect-oriented Software Testing

Empirical studies are becoming more important and required in validating theories in software engineering. In the field of software testing, empirical studies have been used extensively. In this section, empirical studies conducted in evaluating testing approaches for aspect-oriented programs are discussed. Lately, detailed empirical evaluation of the aspect-oriented testing approaches is getting more attention from researchers that was nearly neglected in earlier research work. Lately, detailed empirical evaluation of the aspect-oriented testing approaches is getting more attention from researchers that was nearly neglected in earlier research work. The empirical studies conducted deal with aspect fault detection capability, practicality or usefulness of the proposed aspect–oriented testing approaches or their associated tools.

The first reported experimental work is in [7], to compare performance of double-phase testing with conventional testing methods in unit testing. Since the number of test cases is growing, detecting redundant test cases is crucial in time consuming. An experimental study looking at this issue is done when Respect [23] approach is compared with a technique based on aspect coverage [4]. An empirical evaluation on Aspectra [22] is also performed in which wrapper mechanism shows Aspectra is effective in providing tool support to generate test input for aspectual branch coverage. In regression testing perspective, the work of [39], is empirically evaluated to compare two regression-test-selection techniques which are Java Interclass Graph technique and AspectJ Inter-module Graph technique. An empirical study on ATDG (Automated Test Data Generation), a search-based testing technique, implemented in EAT is performed to compare its performance with random testing [54]. A more recent and thorough empirical analysis to evaluate fault detection effectiveness and test effort efficiency of the four existing automated AOP testing approaches (namely Wrasp, Aspectra, Respect, and EAT) has been performed by [77].

Empirical evaluation for model-based testing of aspect-oriented programs can be found in [41] [71] [72] in which experiments using MACT framework are performed. The empirical study in [41] shows that state-based approach is effective in detecting aspect faults. In addition, another empirical study [71] shows structure-oriented and property-oriented testing strategies complement each other in detecting aspect faults. Subsequent empirical study in this approach investigates the effectiveness of prioritization of transition coverage and round-trip strategies in reporting failure [72].

In the perspective of the practicality or usefulness of approaches or tools, empirical studies conducted show

that structural pairwise approach [63] is practical and useful in integration of object-oriented and aspect-oriented programs, pointcut-based coverage analysis based on *PCDU* [74] is capable to locate related faults, automated generation of pointcut mutants framework [45] is valuable assistance to generate effective mutants, suitability of AjMutator tool [57], AdviceTracer tool [58], MuAspectJ tool [62], and *Proteum/AJ* [67] in mutation analysis. AdviceTracer tool is also empirically compared with JUnit in evaluation of its ability to detect faults in pointcut descriptor [75]. Another empirical study based on UML design models reveals that model-based testing approach is capable of locating aspect-specific faults related to advice and pointcut [47].

As with any empirical study setting in software engineering, especially in software testing, a set of subject programs (benchmark programs) to collect data is needed. For the aforementioned empirical studies in area of AOP, a range of 1 to 14 subject programs (mostly written in AspectJ), from small to big size are used (see Table I).

NonNegative, NonNegativeArg, PushCount, NullCheck, NullChecker, and Instrumentation are parts of Stack program. BusinessRuleImpl is a part of banking program. Most of the subject programs are relatively small in size except iBATIS, Health Watcher, and Tool System Demonstrator. Most of the subject programs are located at:

TABLE I.

LIST OF SUBJECT PROGRAMS USED IN EMPIRICAL STUDIES

Reference	# of Subject Programs	Subject Programs
[47]	1	Greeting card purchase
[62]	1	Health Watcher
[67]	1	Telecom
[75]	1	Health Watcher
[57]	2	Health Watcher, Auction
[58]	2	Tetris, Auction
[72]	2	Telecom, Cruise Control
[41]	3	Telecom, Cruise Control, Banking
[45]	4	Bean, NullCheck, Tetris, Cona-sim
[71]	4	Telecom, Cruise Control, Banking, EJBComponents
[39]	7	Bean, Tracing, Telecom, Quicksort, NullCheck, DCM, LOD
[62]	7	Stack, Subj-obs, Music, Bean, Shape, Point, Telecom
[77]	7	Figure, Bean, Telecom, ProdLine, LOD, NullCheck, DCM
[22]	12	NonNegative,Bean, Telecom, PushCount, NonNegativeArg, Instrumentation, BusinessRuleImpl, StateDesignPattern ProdLine, LOD, NullCheck, DCM
[23]	12	As the above
[74]	12	Bean, Stack, Shape, Subj-obs, Banking, Telecom, Payroll, Music1, Music2, iBATIS, Health Watcher, Toll System Demonstartor
[54]	14	Figure, PushCount, Instrumentation, Hello, Quicksort, NonNegative, NullCheck, NullChecker, Telecom, SavingsAccount, QueueState, ProdLine, DCM, LOD

- http://www.sable.mcgill.ca/benchmarks/
- http://www.eclipse.org/aspectj/doc/released /progguide/index.html
- iBATIS is at http://ibatis.apache.org/index.html
- Health Watcher is at http://www.comp.lancs.ac.uk/~greenwop /tao/implementation.htm
- Toll System Demonstrator is at http://www.aosd-europe.net
- Banking system is in [87]
- Stack program is in [89]

IX. CONCLUDING REMARKS

In this paper, a broad survey of literature on aspect-oriented software testing is given. The topics covered are issues faced, fault models and types, testing coverage, testing techniques, automated support, and empirical studies conducted. For each topic a list of relevant references is given. The references themselves are fully

annotated with a summary of the important findings discussed in each reference.

In the following tables, a brief analysis of the papers referenced is provided. Table II shows how many papers have been published each year. Table III provides an analysis of where the papers have been published.

TABLE II.

ANALYSIS OF CITED PAPERS BY YEAR

Year	*Publications*
2002	1
2003	1
2004	5
2005	14
2006	11
2007	13
2008	9
2009	12
2010	9
2011	6
Total	**81**

TABLE III.

ANALYSIS OF CITED PAPERS BY CATEGORY OF PUBLICATION

Category of Publication	Publications
Journal	
- ISI impact factor	8
- Non-ISI impact factor	4
Conference/Workshop proceedings	58
Technical Report	6
Thesis	3
LNCS/Sigsoft Notes	2

With what said and discussed, we believe the overview of the work related to entire field of AOP testing presented in this paper can help the researcher or practitioner, who is relatively new, in gathering information on the subject and also provide further avenues of exploration for interested researchers.

REFERENCES

[1] J. Zhao. Tool support for unit testing of aspect-oriented software. In *Workshop on Tools for Aspect-Oriented Software Development*, Seattle, Washington, USA, 2002.

Presents a data flow based unit testing approach and its tool support for AOP. It constructs the framed control flow graph to compute def-use pair of the class and aspect, and uses the information to perform data flow testing. The later version of the work is in [3].

[2] R. T. Alexander, J. M. Bieman, and A. A. Andrews. Towards the systematic testing of aspect-oriented programs. *Technical Report CS-4-105*, Colorado State University, 2004.

Introduces a candidate fault model for AOPs and derives testing criteria from the candidate fault model. The fault model is based on interactions that are unique to AOPs. The authors' identify four sources of faults. They are: 1) faults reside in the core concerns and unaffected by aspects, 2) faults in aspect, independent from the woven context, 3) faults emerge when one aspect interacts with a primary abstraction, and 4) faults emerge when one or more aspect woven together into a primary abstraction. Six fault types are proposed: incorrect strength in pointcut patterns, incorrect aspect precedence, failure to establish expected postconditions, failure to preserve state invariants, incorrect focus of control flow, and incorrect changes in control dependencies. This paper is an initial study in developing an effective approach to the systematic testing of AOPs.

[3] J. Zhao. Data-flow-based unit testing of aspect-oriented programs. In *Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC'03),* page 188-197, Dallas, Texas, December 2003.

Proposes a data-flow-based unit testing approach for AOPs by combining unit testing and data flow testing technique to test aspects and classes that may be affected by one or more aspects. The approach performs three levels of unit testing; intra-module, inter-module, and intra-aspect or intra-class. These levels of testing can handle testing problems that are unique to AOPs. The technique proposes a structure model to capture artifacts for testing. This model uses control flow graphs to compute def-use pair of an aspect or a class being tested. The information from the computation is used for selection of tests.

[4] Y. Zhou, H. Ziv, and D. Richardson. Towards a practical approach to test aspect-oriented software. In *Proceedings of 2004 Workshop on Testing Component-based Systems (TECOS 2004)*, September 2004.

This paper proposes an approach to test aspect-oriented software by adapting traditional unit testing, integration testing, and system testing. However, it is an initial approach towards a comprehensive approach to effectively test aspects. The initial approach consists of four steps. The first step deals with developing and testing regular classes in order to isolate and eliminate non aspect-related errors. The second steps concerns with separately woven aspects with regular classes, and the resulting woven aspect is tested for its behavior. In the third step, multiple aspects are woven together with classes to form a complex application. In this step several aspects are integrated and tested. In the final step, all aspects are woven together with regular classes as a complete system. The complete system is then tested. In this paper, testing coverage is defined for how well an aspect is tested by a set of test cases. The paper suggests reusing test cases used for testing regular classes for testing aspects as a means to reduce testing cost. An algorithm for selecting relevant test cases and calculating testing coverage is also proposed. A Java tool is developed in the Eclipse environment for this approach.

[5] M. Mortensen and R. T. Alexander. Adequate testing of aspect-oriented programs. *Technical Report CS04-110*, Department of Computer Science, Colorado State University, Fort Collins, Colorado, USA, December 2004.

Presents a fault-based testing framework consisting of coverage criteria and mutation testing to define adequate testing of AOPs. Coverage criteria applied on aspect structure are statement coverage, insertion coverage, context coverage, and def-use coverage. Meanwhile mutation operators used are pointcut strengthening, pointcut weakening and precedence changing. The paper demonstrates the application and benefits gained by using this approach. The authors identify the need for an integrated tool to test AspectJ programs.

[6] W. Xu, D. Xu, V. Goel and K. Nygard. Aspect flow graph for testing aspect-oriented programs. http://www.cs.ndsu.nodak.edu/`wxu/research/436-1111jd.pdf. 2004.

Presents a hybrid testing model approach that combines responsibility-based testing model and an implementation-based testing model. The paper shows that the test suits generated by the model are manageable, code based and executable. The approach taken is merging class state model and aspect state model into aspect scope state model (ASSM). An aspect flow graph (AFG) is generated by combining ASSM with advice and method data flow graph. Based on the AFG and the transition tree created, concert and executable code-based test suites can be derived in terms of coverage testing criteria.

[7] G. Xu, Z. Yang, H. Huang, Q. Chen, L. Chen, and F. Xu. JAOUT: Automated generation of aspect-oriented unit test. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04),* Shanghai, China. 2004.

Proposes a framework for automating the unit test generation and test oracles from aspects in AOP. The approach taken is based on a new concept known as application-specific aspect. In this concept, aspects for the same use are gathered into application-related top-level aspects. The approach makes use of Aspect-Oriented Test Description Language (AOTDL) to specify the testing aspects. This then is translated by their proposed tool, JAOUT/translator, into common aspects in AspectJ. The tool JAOUT/generator generates the test classes (serve as test oracles) for JUnit from the AspectJ programs. Then, the authors use double-phase testing to filter out the meaningless test cases.

[8] D. Xu, W. Xu and K. Nygard. A State-based approach to testing aspect-oriented programs. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'05),* July 14-16, Taiwan. 2005

Proposes an approach to generate test suites for adequately testing object behavior and interaction between classes and aspects in terms of message sequences. The paper presents a model called Aspectual State Model (ASM), an extension of FREE (Flattened Regular Expression) state model, to specify classes and aspects. ASM represents state transition in AOP. The approach taken is by transforming an ASM to a transition tree. In the transition tree, each path from the root to a terminal leaf node is a test requirement (message sequence). However, the ASM is defined in an ad hoc manner.

[9] W. Xu and D. Xu. A model-based approach to test generation for aspect-oriented programs. *AOSD2005 Workshop on Testing Aspect-Oriented Programs,* Chicago, USA, March 2005.

Presents an approach to generate tests that are adequately testing interaction between classes and aspects based on aspect-oriented UML models. The approach makes use of the extension to UML models. These models are class diagrams, aspect diagrams, and sequence diagrams. The approach firstly weaves advices on a particular method into a new sequence diagram (woven sequence diagram). Secondly, by using goal-oriented reasoning on the woven sequence diagram, a goal-directed flow graph is generated for certain coverage criteria. This paper uses polymorphic and branch coverage. Thirdly, the flow graph is transformed to a flow tree. Each path from a leaf to the root in the flow tree is a sequence of requested messages or method invocations thus, indicates a test case. The paper provides the related algorithms for each step taken in the approach. However, the paper is the authors' preliminary report on their ongoing research on model-based testing of AOPs.

[10] C. V. Lopes and T. C. Ngo. Unit-testing aspectual behavior. In *Proceedings of the Workshop on Testing Aspect-Oriented Programs,* Chicago, USA, March 2005 *This is a position paper to find a solution for unit-testing aspects. The approach taken is based on JAML (Java Aspect Markup Language) and JamlUnit (an extension of JUnit). JamlUnit is a framework for performing unit testing of aspect written in JAML. The paper shows the possibility and requirements for formulating clean unit testing techniques using mock object mechanisms.*

[11] M. Mortensen and R. T. Alexander. An Approach for adequate testing of AspectJ Programs. In *Proceedings of the 1st Workshop on Testing Aspect-Oriented Programs –* in Conjunction with AOSD'2005, Chicago, USA, 14-18 March 2005

Discusses an approach similar to [5], but with lesser number of applications.

[12] H. Rajan and K. Sullivan. Generalizing AOP for aspect-oriented testing. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'2005)*, Chicago, USA, 14-18 March 2005.

Proposes a language-centric approach to automate test adequacy analysis. The approach is called concern coverage. The idea in this approach is to make use of declarative aspect-oriented pointcut constructs to include the code in adequacy criteria. They use the idea of generalized join point models and pointcut languages, and generalized advices. These ideas are used to extend Eos language to become Eos-T. A framework embodies the ideas is provided to support a testing tool, AspectCov.

[13] S. A. Ali Naqvi, S. Ali, and M. Uzair Khan. An evaluation of aspect oriented testing techniques. In *Proceedings of the 2005 International Conference on Emerging Technologies*, Islamabad, 17-18 September 2005.

Presents an evaluation of data-flow-based unit testing approach [3], aspect flow graph based testing strategy [6], and state-based testing strategy [8] with respect to fault model proposed by [2]. The authors found out that state-based testing strategy of AOPs may produce good results.

[14] T. Xie, J. Z, D. Marinov, and D. Notkin. Automated test generation for AspectJ programs. *The 1st Workshop on Testing Aspect-Oriented Programs*, 2005.

Proposes a framework, Wrasp, to reduce the manual testing effort by automating generation of both unit and integration tests for AspectJ Programs. Wrasp is developed by leveraging the existing Java test generation tools such as Parasoft Jtest and JCrasher. For integration testing, Wrasp synthesizes a wrapper class for base class and then feeds it the existing test generation tools. Wrasp takes care of integration of advised methods, advice, and intertype methods. For unit testing, Wrasp generates tests to test advice in isolation.

[15] M. Badri, L. Badri, and M. Bourque-Fortin. Generating unit test sequences for aspect-oriented programs: towards a formal approach using UML state diagrams. In *Proceedings of the 3rd International Conference on Information and Communication Technology*, Cairo, 5-6 December 2005, pp. 237-253.

Presents a technique for unit test of AOPs based on dynamic behavior as described by UML statecharts. The technique taken generates test sequences from an extended statechart that integrates aspects to the classes. The test sequences generated in accordance to new testing criteria that suit the aspectual behaviors. The strategy of the technique focuses on finding faults in integration of aspects.

[16] Y. Yamazaki, K. Sakurai, S. Matsuura, H. Masuhara, H. Hashiura, and S. Komiya. A unit testing framework for aspects without weaving. In *Proceedings of the 1st Workshop on Testing Aspect-Oriented Programs,* March 2005.

Proposes a framework for unit test without weaving an aspect by describing test cases viewed as description of aspect in the program. Thus, test cases can directly verify aspects' properties such as the advice behavior and pointcut matching. This framework is possible to be combined with other framework written in Java through the use of APIs.

[17] M. Ceccato, P. Tonella and F. Ricca. Is AOP code easier or harder to test than OOP code? In *Proceedings of the Workshop on Testing Aspect-Oriented Programs,* Chicago, USA, March 2005.

Extends faults type in [2] with incorrect changes in exceptional control flow, failures due to inter-type declarations, and incorrect changes in polymorphic calls. Based on the fault types, this paper describes the possibility of using incremental testing to allow separately testing the base code and aspects in successive steps.

[18] A. V. Deursen, M. Marin, and L. Moonen. A systematic aspect-oriented refactoring and testing strategy, and its application to JHotDraw. arXiv:cs/0503015v1 [cs.SE], March 2005.

Proposes a strategy for adoption of aspect-orientation in existing software through refactoring and testing. Refactoring will result in improvement of the internal structure of a software system without altering its behavior. The proposed testing strategy, aims to ensure consistent migration process, consists of an aspect-oriented fault model and adequacy criteria. The faults covered are due to inter-type declarations, faults in pointcuts and faults in advice. The strategy is implemented in an open source project JHotDraw.

[19] P. Massicotte, L. Badri, and M. Badri. Generating aspects-classes integration testing sequences: a collaboration diagram based strategy. In *Proceedings of the 3rd ACIS International Conference on Software Engineering Research, Management and Applications (SERA'05),* pp. 30-37, August 2005.

Proposes a technique to generate test sequences that are based on dynamic interaction between aspects and classes for certain testing criteria. The technique focuses on handling the integration of one or more aspects in collaboration with a group of objects specified using UML collaboration diagrams. The testing criteria are: transition coverage criterion, sequence coverage criterion, modified sequences coverage criterion, simple integration coverage criterion and multi-aspects integration coverage criterion.

[20] P. Massicotte, L. Badri, and M. Badri. Aspects-classes integration testing strategy: an incremental approach. 2nd International Workshop on Rapid Integration of Software Engineering Techniques (*RISE 2005), LNCS 3943*, 2005, pp. 158-173.

Presents an approach based on specifications described in a collaboration diagram. The approach consists of two phases. The first phase uses static analysis to generate test sequences based on interactions between aspects and classes. The second phase verifies the execution of the generated sequences in which aspects are incrementally integrated to the collaboration diagrams. The testing criteria as [19] are used.

[21] D. Xu and W. Xu. State-based incremental testing of aspect-oriented programs. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06),* March 20-24, pg. 180-189, Bonn, Germany, 2006.

This paper is a follow up to the authors' previous paper. This paper presents a state-based approach to incrementally test AOPs. The approach taken is by incrementally modifying aspects to their base classes. It describes two perspectives. Firstly, formalizes the definition of the aspect-oriented state model by extending the traditional finite state model. Besides that weaving mechanism for applying an aspect to a base model is also defined. These facilitate the transformation of transition tree and generation of abstract test cases for both base classes and aspect-oriented programs. Secondly, the investigation of reusing base class tests

reveals that majority of base class tests can be reused for aspects. Several rules for maximizing reuse of base class tests for aspect are identified. However, slight modifications to the base class tests are necessary. Besides the two perspectives, the paper also describes several types of aspect-specific faults that can be revealed by the state-based testing.

[22] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of AspectJ programs. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06)*, March 20-24, pg. 190-201, Bonn, Germany, 2006.

Presents a framework known as Aspectra to automate generation of test inputs for testing aspects in AspectJ. The framework utilizes a wrapper-synthesis technique to prepare woven classes for test-generation tools, such as Rostra and Symstra, based on state exploration. This approach uses the same process as [14]. To assess the quality of the generated test input, aspectual branch coverage and interaction coverage are defined and measured. Aspectra has been applied on 12 AspectJ benchmark source code. Their results provide useful guidance to improve test coverage.

[23] T. Xie, J. Zhao, D. Marinov, and D. Notkin. Detecting redundant unit tests for AspectJ Programs. In *Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE'06)*, Raleigh, NCs, USA, 2006.

Proposes a framework known as Raspect for detecting redundancy in unit testing of AspectJ programs. It is a complementary to Wrasp [14] and Aspectra [22]. Raspect uses partly the same process as in Wrasp and Aspectra. The difference is where Raspect minimizes the generated test for detecting and removing redundant. Three levels of units in AspectJ programs are introduced and tested: advised method, advice, and intertype methods. Raspect is an extension of Rostra which detects redundant test for Java methods. Raspect is evaluated against [4] and the results show that Raspect can effectively reduce the size of generated test suites.

[24] P. Anbalagan and T. Xie. APTE: Automated pointcut testing for AspectJ programs. In *Proceedings of the 2nd Workshop on Testing Aspect-Oriented Programs*, pages 27-32, July 2006.

An automated framework to unit test pointcuts in AspectJ programs with the help of existing framework that perform unit testing without weaving[16]. This framework identifies joinpoints that are matched with a pointcut expression and a set of boundary joinpoints. The boundary joinpoints are events that do not satisfy a pointcut expression but are close to the matched joinpoints. The boundary joinpoints are identified as those unmatched joinpoint candidates whose distances from the match joinpoints are less than a predefined threshold value. The threshold value is the maximum distance against which the distances of unmatched joinpoint candidates are compared and is supplied by the user of the framework. The distance measured quantifies the deviation of the boundary joinpoint from the matched ones. The measure used is calculated based on the Levenshtein algorithm

[25] P. Anbalagan and T. Xie. Efficient mutant generation testing of pointcuts in Aspect-oriented programs. In *Proceedings of the 2nd workshop on Mutation Analysis, 2006.*

Proposes a framework that automatically generates mutants for a pointcut expression and identifies the mutants that are closely resemble the original expression (equivalent mutants). The processes taken are identifying join points that are matched by a pointcut expression, generating mutants for this pointcut, and indentifying join points matched by the mutants. Then the mutants and their matched join points are compared with their original pointcut. The mutants then are classified into the same set of join points as neutral, weak, or strong. The best mutant for a particular set is selected using a simple heuristics. The classified mutants are ranked to help developer in choosing a mutant that resembles closely the original one. The ranking is based on a string similarity measure- Monge Elkan distance measure. The framework is able to reduce the total number of mutants from the initial large number of generated mutants. An extended version of this paper can be found in [45].

[26] O. A. L. Lemos, F. C. Ferrari, P. C. Masiero, and C. V. Lopes. Testing aspect-oriented programming pointcut descriptors. In *Proceedings of the 2nd Workshop on Testing Aspect-Oriented Programs*, pages 33-38, July 2006.

Presents a fault classification for pointcut descriptorsas introduced by [2] and a two-step strategy in handling the fault introduced in by pointcut descriptors. The strategy is a) detecting extra join points selected by by pointcut descriptors using structural testing, and b) detecting intended join points that were not selected by pointcut descriptors using mutation testing.

[27] J. Zhao, T. Xie, and N. Li. Towards regression test selection for AspectJ Programs. In *Proceedings of the 2nd Workshop on Testing Aspect-Oriented Programs*, pages 21-26, July 2006.

Presents a code-base technique to safely regression test AspectJ programs. The technique makes use of control flow graphs of an original AspectJ program and its modified version to detect the dangerous arcs. In order to facilitate selection of regression tests, a control flow graph that captures information on aspects is proposed.

[28] G. Xu. A regression tests selection technique for aspect-oriented programs. In *Proceedings of the 2nd Workshop on Testing Aspect-Oriented Programs*, pages 15-20, July 2006.
This position paper defines a new test selection criterion based on impact of an aspect on main classes for AOPs to achieve higher precision. The three-phase technique used to safely selects regression tests based on the differences of control flow paths of two programs and uses dynamic analysis to re-select the tests using the new criterion.

[29] J. S. Baekken and R. T. Alexander. Towards a fault model for AspectJ programs: Step 1 – pointcut faults. In *Proceedings of the 2nd Workshop on Testing Aspect-Oriented Programs*, pages 1-6, July 2006.

Presents a fault model for AspectJ pointcut by describing it through format that contains fault name, fault category, summary, syntactic form and semantic impact. It identifies four pointcut fault categories: incorrect patterns, incorrect choice of primitive pointcut, incorrect matching of dynamic circumstances, and incorrect pointcut composition.

[30] J. S. Baekken and R. T. Alexander. A candidate fault model for AspectJ pointcuts. In *Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE'06)*, Raleigh, NCs, USA, pg. 169-178, 2006.

Describes in detail the individual fault types in each category identified in [29].

[31] J. S. Baekken. A fault model for pointcuts and advice in AspectJ programs. Master's thesis, School of Electrical Engineering and Computer Science, Washington State Univ., USA, 2006.

Presents a fault model for pointcuts and advice of the AspectJ programming language. The fault model for pointcuts has been discussed in [29] and [30]. The thesis provides a fault/failure analysis in the form of how a fault found in a pointcut or a piece of advice can cause a data state in the program to become corrupted, and how that erroneous data state can propagate to the failure of the program. Catalog of fault types is identified for the fault model. Each type of fault is described in terms of how it appears syntactically in source code and how it can cause an infection of program state.

[32] O. A. L. Lemos, A. m. R. Vincenzi, J. C. Maldonado, and P. C. Masiero. Control and data flow structural testing criteria for aspect-oriented programs. The Journal of Systems and Software. Vol. 80, issue 6, pp. 862-882, 2007.

Proposes a derivation of a control and data flow model, named as aspect-oriented def-use graph (AODU), of AspectJ programs based on static analysis of the Java bytecode. From AODU , control flow and data flow based testing criteria for aspect-oriented programs are defined. This model is used to support structural testing to unit testing of AspectJ programs.

[33] T. Xie and J. Zhao. Perspective on automated testing of aspect-oriented programs. In *Proceedings of the 3rd Workshop on Testing Aspect-Oriented Programs*, March 12-13, 2007, Vancouver, British Columbia, Canada.

This position paper presents the authors view on automated testing of aspect-oriented programs. Their views are classified on three dimensions: testing aspectual behavior or aspectual composition, unit tests or integration tests, and test-input generation or test oracles. The paper presents the techniques for test-input generation that is based on wrapper mechanism [22] to leverage the existing Java testing tool for AspectJ programs, test selection based on coverage information [3], [22], [23] and mutation testing [25], and runtime behavior checking with specification.

[34] C. Zhao and R. Alexander. Testing AspectJ programs using fault-based testing. In *Proceedings of the 3rd Workshop on Testing Aspect-Oriented Programs*, pp. 13-16, March 12-13, 2007, Vancouver, British Columbia, Canada.

This position paper describes an AspectJ program testing method based on fault model. The fault model is derived with the support of dependency model and interaction model.

[35] C. Zhao and R. Alexander. Testing aspect-oriented programs as object-oriented programs. In *Proceedings of the 3rd Workshop on Testing Aspect-Oriented Programs*, pp. 23-27, March 12-13, 2007, Vancouver, British Columbia, Canada.
Discusses the feasibility of testing the woven code using an object-oriented programs testing method.

[36] P. Massicotte, L. Badri and M. Badri. Towards a tool supporting integration testing of aspect-oriented programs. Journal of Object Technology, Vol. 6, No. 1, pp. 67-89, January-February 2007.

Presents an aspects-classes integration testing strategy and its associated tool for testing AspectJ programs. The approach taken in this strategy consists of a) generating test sequences based on the dynamic interaction between aspects and classes, and b) verifying the execution of the selected sequences. This paper is an extension to the work in [19] and [20].

[37] M. L. Bernardi and G. A. Di Lucca. Testing aspect oriented programs: an approach based on the coverage of the interactions among advices and methods. In *Proceedings of the 6th International Conference on the Quality of Information and Communication Technology*, pp. 65-76, 2007.

Proposes a set of testing coverage criteria based on the interactions among the advices and the methods. A fault model is proposed based on the interactions of methods and advice by means of pointcut expressions defining join points. The authors make use of the Inter-procedural Aspect Control Flow Graph that they have previously developed to define the coverage criteria. The criteria are adapted from traditional white-box unit testing. The criteria provide a guideline to define test cases when the control is passed from classes to aspects, and when a possible fault is likely to occur.

[38] R. M. Parizi and A. A. Ghani. A survey on aspect-oriented testing approaches. In *Proceedings of the 5th International Conference on Computational Science and Applications*, pp. 78-85, 2007.

Surveys and compare the effectiveness the testing approaches on AOP that are data flow-based [3], state-based [8], aspect flow graph [6], aspectual behavior [10], and model-based [9]. This is done in terms their ability to find different kind of faults [2].

[39] G. Xu and A. Rountev. Regression Test Selection for AspectJ software. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pp. 65-74, 2007.

Proposes a control flow representation of the semantics of AspectJ source code known as AspectJ Inter-module Graph (AJIG) for regression testing of AOPs. The graph consists of: a) CFGs that model the control flow for Java classes, aspects, and relationships between aspects and classes through non-advice method calls, and b) interaction graphs that model the interactions between methods and advices at some specific join points. Two-phase graph traversal algorithm is developed to identify the differences between two versions of AspectJ programs. This approach claims to reduce the number of test case selection.

[40] S. Zhang and J. Zhao. On identifying bug patterns in aspect-oriented programs. In *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, Beijing, pp. 431-438, 24-27 July 2007.

Identifies six bug patterns in AspectJ programming language. The patterns are The Infinite Loop, The Scope of Advice, The Multiple Advice Invocation, The Unmatched Join Point, Misuse of getTarget(), and Introduction Interference. Examples are shown for each bug patterns. The patterns can be helpful for testing.

[41] W. Xu. Testing aspect-oriented programs with state models. *PhD Dissertation*, North Dakota State University of Agriculture and Applied Science, May 2007.

Presents a state-based approach in modeling and testing of aspect-oriented programs. The approach taken is by using extended finite state machines to model classes and aspects. Some related rules concerning the impacts of aspects impose on the state transitions of the base class objects are defined. These are used to compose aspect models together with their base class models through weaving process. An incremental testing process is adopted to locate failures. A series of experiments with a number of mutants are conducted to evaluate the fault-detection ability of the approach.

[42] D. Xu and X. He. Generation of test requirements from aspectual use case. In *Proceedings of the 3rd Workshop on Testing Aspect-Oriented Programs*, pp. 17-22, March 12-13, 2007, Vancouver, British Columbia, Canada.

Presents an approach to generate system test requirements from aspect-oriented use cases. This is done through formalization of a testable system model from aspect-oriented use cases. The steps taken are 1) transforming aspect-oriented use case diagram and description into aspect-oriented Petri nets, 2) traversing the woven Petri net according to test coverage criteria to generate use case sequences. The criteria are use case coverage, transition coverage, and state coverage.

[43] I. G. Franchin, O. A. L. Lemos, and P. C. Masiero. Pairwise structural testing of object and aspect-oriented Java programs. In *Proceedings of the 21st Brazilian Symposium on Software Engineering*, pp. 377-393, Porto Alegre, RS, Brasil. SBC Press. 2007.

Presents a structural integration testing approach for object-oriented and aspect-oriented Java programs. This is proposed to handle the testing of the interaction between pairs of units (methods and advices) with respect to the correctness of their interface. Based on Java bytecode data-flow and control-flow model, a model known as PWDU (PairWise Def-Use) graph is proposed to represent the control and data-flow of pairs of units. From PWDU, three testing criteria are defined: all-pairwise-integrated-nodes, all-pairwise-integrated-edges, and all-pairwise-integrated-uses. The approach is implemented in JaBUTi/PW-AJ an extension of JaBUTi (Java Bytecode Understanding and Testing) family of testing tool.

[44] F. Wedyan and S. Ghosh. A joinpoint coverage measurement tool for evaluating the effectiveness of test inputs for AspectJ programs. In *Proceedings of the 19th International Symposium on Software Reliability Engineering*, pp. 207-212, 10-14 Nov, 2008.

Presents a tool for measuring joinpoint coverage for AspectJ programs. The approach taken by the tool is by analyzing the bytecode of the woven classes and aspects to get joinpoint information. The coverage measures considered are the coverage in the woven classes and the coverage for the advice.

[45] P. Anbalagan and T. Xie. Automated generation of pointcut mutants for testing pointcuts in AspectJ programs. In *Proceedings of the 19th International Symposium on Software Reliability Engineering*, pp. 239-248, 10-14 Nov, 2008.

This is an extended version of the author's position paper [25]. More detail discussion on the automated framework is given. The authors also presents the results of the empirical study conducted. The results show that the framework is valuable to assist in generating effective mutants.

[46] F. C. Ferrari, J. C. Maldonado, and A. Rashid. Mutation testing for aspect-oriented programs. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation*. pp. 52-61, 9-11 April 2008.

Presents a set of mutation operators for mutation testing of AspectJ programs. The set is design to model fault instances of fault types found in [2], [17], [18], [26], [31], [40]. This paper groups the fault types and respective operators into related language features. The operators can be used for other AspectJ-like programs.

[47] D. Xu, W. Xu, and W. E. Wong. Testing aspect-oriented programs with UML design models. International Journal of Software Engineering and Knowledge Engineering, Vol. 18, No. 3, pp. 413-437, 2008.

This is an improved version of paper [9] from the perspectives of aspect-oriented UML models, model-based test generation, and test execution. The paper also discusses the empirical study conducted and the results show that model-based testing approach able to reveal fault types such as incorrect advice type, incorrect pointcut strengths, and incorrect aspect precedence.

[48] M. Amar and K. Shabbir. Systematic review on testing aspect-oriented programs: Challenges, techniques and their effectiveness. *Master Thesis Software Engineering*, School of Engineering, Blekinge Institute of Technology, Sweden, August 2008.

Presents a systematic review of aspect-oriented software testing from the perspectives of challenges, techniques, and their effectiveness. The review provides detail state-of-the-art research on aspect-oriented software testing from the year 1999 to 2008. The review focuses on structural testing techniques.

[49] C. H. Liu and C. W. Chang. A state-based testing approach for aspect-oriented programming. Journal of Information Science and Engineering, Vol. 24, pp. 11-31, 2008.

Presents a state-based testing approach for AOP. The approach considers changes in the state-based behavior introduced by the advices in different aspects. Three steps are taken by the approach in generating test cases. Firstly, identify the state variables and transition from AOP specification or source code. This is done with the help of object state diagram (OSD) to represent the state-based behavior of AOP program before aspect weaving, crosscut weaving model (CWM) to abstract weaving sequences and to analyze the changes of state variables after advice weaving. Secondly, capture the possible transition changes caused by the aspects. This is done with the help of an aspect object state diagram (AsOSD). Lastly based on AsOSD, construct a test tree by integrating the transitions of state variables.

[50] M. Bernardi. Reverse engineering of aspect oriented systems to support their comprehension, evolution, testing and assessment. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR 2008)*, pp. 290-293, 1-4 April, Athens, 2008.

Presents a structural testing approach based on an inter-procedural aspect control flow graph (IACFG). The IACFG represents the interactions aspects and OO component of AO system. The IACFG model together with defined set of coverage criteria are used for testing AO programs.

[51] O. A. L. Lemos and P. C. Masiero. Integration testing of aspect-oriented programs: A structural pointcut-based approach. In *Proceedings of the 22nd Brazilian Symposium on Software Engineering*, pp. 49-64, 2008.

Presents a structural integration testing approach for AspectJ programs. This is proposed to test pieces of advice at each join point in the AO programs with respect to pointcut mechanisms. Based on the idea of pairwise approach [43], A model known as Pointcut-based Control Flow Graph (PCCFG) is defined to represent the flow of control between the base units and pieces of advice (execution regions of AO programs affected by a pointcut). Two control flow criteria are defined: all-pointcut-based-advice-nodes and all-pointcut-advice-edges. The approach is implemented in JaBUTi/PC-AJ tool.

[52] O. A. L. Lemos and P. C. Masiero. Using structural testing to identify unintended join points selected by pointcuts in aspect-oriented programs. In *Proceedings of the 32nd Annual IEEE Software Engineering Workshop (SEW 2008),* pp. 84-93, 15-16 Oct, 2008.

Presents an approach using integration structural testing to test unintended join point caused by faulty pointcuts. The contents are basically similar to [51].

[53] C. Babu and H. R. Krishnan. Fault model and test-case generation for the composition of aspects. SIGSOFT Software Engineering Notes, Vol. 34, No. 1, pp1-6, 2009.

Proposes a fault model that identifies the faults occurred during aspect composition from the perspective of design based on sequence diagrams. Test case generation is done based on the sequence diagrams. Possible faults that may occur are incorrect aspect precedence, incorrect focus of control flow between aspects and classes, violation of the conditional order of execution of aspects, and incorrect focus of control flow between aspects.

[54] M. Harman, F. Islam, T. Xie, and S. Wappler. Automated test data generation for aspect-oriented programs. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development (AOSD'09),* March 2-6, pg. 185-196, Charlottesville, Virginia, 2009.

Introduces a search-based optimization approach to generate test data automatically for structural coverage of aspect-oriented programs. Specifically an approach is developed to generate test data to cover aspectual branches (branches inside aspects) based on evolutionary testing, a search-based software testing approach. Input-domain reduction technique is used together with program slicing technique to reduce the input domain by excluding irrelevant parameter in the search space to reduce test effort and increase testing effectiveness. The approach is implemented in a prototype tool known as EvolutionaryAspectTester (EAT). Empirical study conducted shows that the optimization used increases the effectiveness and efficiency by focusing on aspectual behavior.

[55] N. Kumar, A. Rathi, D. Sosale, and S. N. Konuganti. Enabling the adoption of aspects – testing aspects: A risk model, fault model and patterns. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development (AOSD'09),* March 2-6, pp. 197-206, Charlottesville, Virginia, 2009.

Presents an AOP testing model that consists of a faulty model, a risk model for assessing related AOP faults, a testing framework that supports unit test and regression test, and AOp testing patterns.

[56] R. M. Parizi, A. A. A. Ghani, R. Abdullah, and R. Atan. Towards a framework for automated random testing of aspect-oriented programs. In *Proceedings of the ISCA 18th International Conference on Software Engineering and Data Engineering (SEDE 2009),* pp. 217-223, 22-24 June, Las Vegas, Nevada, USA, 2009.

This position paper proposes a framework to random test aspect-oriented programs. The aim is to combine random testing with AOP testing in automating the test data generation and execution of testing for aspect-oriented programs.

[57] R. Delamare, B. Baudry, and Y. Le Traon. AjMutator: A tool for the mutation analysis of AspectJ pointcut descriptors. In *Proceedings of the IEEE International Conference on Software Testing Verification and Validation Workshops (ICSTW 2009)*, pp. 200-204, 1-4 April, Denver, Colorado, 2009.

Presents a tool known as AjMutator for mutation analysis of pointcut descriptor (PCD) in AspectJ programs. Mutation operators for PCDs [46] are considered in the tool. The tool also automatically classifies the mutant PCDs according to matching of sets of jointpoints with the mutants to look for equivalent mutants. The mutant classification is divided into equivalent mutant, mutant with neglected jointpoint and mutant with unintended jointpoit.

[58] R. Delamare, B. Baudry, S. Ghosh and Y. Le Traon. A test-driven approach to developing pointcut descriptors in AspectJ. In *Proceedings of the 2009 International Conference on Software Testing, Verification and Validation (ICST '09)*, pp. 376-385, 1-4 April, Denver Colorado, 2009.

Identifies two issues related to testing pointcut descriptors (PCDs) that lack of specification for intended joinpoints and inability of JUnit to explicitly assert the presence or absence of an advice at a given point in source code. Thus this paper proposes a test-driven approach to solve the issues. A tool known as AdviceTracer that extended JUnit is implemented to specify the expected joinpoints. AjMutator [57] is used to inject fault into PCDs in order to help validating test cases produced by AdviceTracer. Experiments conducted shows that AdviceTracer works better in test-driven development of PCDs better than plain JUnit.

[59] M. Kumar, A. Sharma, and S. Garg. A study of aspect oriented testing techniques. In *Proceedings of the IEEE Symposium on Industrial Electronics and Applications (ISIEA 2009)*, pp. 996-1001, Oct 4-6, Kuala Lumpur, Malaysia, 2009.

Analyzes the effectiveness of four testing strategies [3], [6], [8], [37] with respect to finding different types of faults as defined in the fault model [2].

[60] M. Badri, L. Badri and M. Bourque-Fortin. Automated state-based unit testing for aspect-oriented programs: A supporting framework. *Journal of Object Technology*, Vol. 8, No. 3, May-June 2009, pp. 121-146.

Presents a state-based unit testing technique, based on UML Statechart Diagrams of the classes under test and the code of related aspect, to integrate one or more aspects to a class in AspectJ programs. The technique focuses on aspect-class block of code. As in a associated tool, AJUnit, is developed to generate testing sequences covering the block of code. These sequences represent different scenarios of the statechart diagram with extension of behavior of related aspects. The tool also supports the execution and verification of the generated sequences.

[61] R. M. Parizi, A. A. A. Ghani, R. Abdullah, and R. Atan. On the applicability of random testing of aspect-oriented programs. International Journal of Software Engineering and Its Applications, Vol. 3, No. 3, July, pp.1-19, 2009.

Presents an extension of the earlier paper [56].

[62] A. Jackson and S. Clarke. MuAspectJ: Mutant generation to support measuring the testability of AspectJ programs. *Technical Report (TCD-CS-2009-38)*, ACM, September 2009.

Introduces MuAspectJ, a tool that generates mutants for AspectJ programs. It is an extension of MuJava. The tool provides a complete set of mutation operators to cover both AO and non-AO locations in AspectJ programs. The operators create faults at the locations in a source code. The tool is evaluated based on the quality of the mutants generated by means of location coverage and mutation density.

[63] O. A. L. Lemos, I. G. Franchin, and P. C. Masiero. Integration testing of object-oriented and aspect-oriented programs: A structural pairwise approach for Java. Science of Computer Programming. Vol. 74, pp. 861-878, 2009.

Presents an extended version of their earlier paper [43], in proposing a structural integration testing approach for object-oriented and aspect-oriented Java programs to handle the testing of the interaction between pairs of units (methods and advices). A model called PWDU (PairWise Def-Use) graph is proposed and used to represent the control and data-flow of pairs of units, and its family of testing criteria. Exploratory evaluation involving experiments conducted to investigate the cost of application and usefulness of the approach shows that the criteria are practical and useful.

[64] F. C. Ferrari, E. N. Hohn, and J. C. Maldonado. Testing aspect-oriented software: Evolution and collaboration through the years. In *Proceedings of LAWASP'09*, Brazilian Computer Society, pp. 24-30, 2009.

Presents a systematic review on general scenario of research on AO software testing focusing on the evolution of AOP testing approaches and collaborations among researchers in AO testing community. A total of 51 studies have been selected from the year 2002 and 2009. The paper shows that structural and fault-based testing strategies have reached the highest maturity level. However, there is still lack of collaboration among researchers.

[65] F. Wedyan and S. Ghosh. A dataflow testing approach for aspect-oriented programs. In *Proceedings of the 2010 IEEE 12th International Symposium on High Assurance Systems Engineering (HASE 2010)*, pp. 64-73, 2010.

Presents a dataflow testing approach for AOPs that is based on class state variables. The approach uses framed Interprocedural Control Flow Graph (ICFG) [3]. Based on class state variables, five types of Definition-Use Association are classified from which six state variable test criteria are proposed. The approach has been implemented in a tool known as DCT-AJ to measure the dataflow coverage for a test suite. Cost-effectiveness studies conducted shows that the dataflow criteria were more effective than block coverage criteria.

[66] F. Ferrari, R. Burrows, O. Lemos, A. Garcia, E. Figueiredo, N. Cacho, F. Lopes, N. Temudo, L. Silva, S. Soares, A. Rashid, P. Msiero, T. Batista, and J. Maldonado. An exploratory study of fault-proneness in evolving aspect-oriented programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, pp. 65-74, 2-8 May, Cape Town, South Africa, 2010.

Presents an exploratory analysis on fault-proneness of AOPs of three real world AO systems from different application domains. The analysis concerns with AOP obliviousness properties and mechanisms that are pointcuts, advices and intertype declarations. The methods used for the analysis are through testing and static analysis. These are performed by developers and independent testers. Firstly, the result confirms that the lack of awareness between base and aspectual modules leads to incorrect implementation. Secondly, the current AOP mechanisms show similar fault-proneness in overall systems and concern specific implementation, not only pointcuts. Lastly, it is found out that there is a direct proportional between the number of faults associated with a concern and the number of AOP mechanisms implementing the concerns.

[67] F. C. Ferrari, A. Rashid, E. Y. Nakagawa, and J. C. Maldonado. Automating the mutation testing of aspect-oriented Java programs. In *Proceedings of the 5th Workshop on Automation of Software Test (AST'10)*, pp. 51-58, 3-4 May, Cape Town, South Africa, 2010.

Introduces a tool known as Proteum/AJ to automate the mutation testing of aspect-oriented AspectJ programs. The tool supports a set of requirements that defines the functionalities identified for mutation testing. The tool is also able to overcome limitation faced by previous tools [45] [57]. The tool implements a set of mutation operators defined in [46].

[68] M. Singh and S. Mishra. Mutant generation for aspect-oriented programs. Indian Journal of Computer Science and Engineering, Vol. 1, No. 4, pp. 409-415, 2010.

Presents an extension to the fault types defined in [46]. Based on the fault types, a set of mutation operators is specified. It also proposes a framework to implement the faults types and mutation operators.

[69] R. M. L. M. Moreira, A. C. R. Paiva, and A. Aguiar. Testing aspect-oriented programs. In *Proceedings of the 5th Iberian Conference on Information Systems and Technologies*, pp. 1-6, 16-19 June, Santiago de Compostela, 2010.

Presents a perspective on software quality issues introduced by AOP, identifies key issues in testing AOP, and reviews state-of-the-art of the proposed solutions.

[70] F. C. Ferrari, R. Burrows, O. A. L. Lemos, A. Garcia, and J. C. Maldonado. Characterising faults in aspect-oriented programs: Towards filling the gap between theory and practice. In *Proceedings of the 2010 Brazilian Symposium on Software Engineering*, pp. 50-59, Sept 27- Oct 1, 2010.

Presents results of a study (using testing and code analysis) that quantify and categorize faults in AOPs. Fine-grained fault taxonomy refined from [46] is proposed in the study. The faults are extracted from three AO systems. The results show that a subset of fault types stood out when compared to faults within a specific category. The results also show the most recurring fault types and how they are introduced to the code.

[71] D. Xu, O. El-Ariss, W. Xu, and L. Wang. Testing aspect-oriented programs with finite machines. Journal of Software Testing, Verification and Reliability, doi: 10.1002/stvr.440, 2010.

Presents MACT (Model-based Aspect Checking and Testing) framework, based on finite state models, for testing the conformance of aspect-oriented programs against their aspect-oriented state models. MACT provides notations for describing aspect-oriented properties at UML state machine. MACT offers structure-oriented and property-oriented testing strategies for generating aspect tests from aspect-oriented state model. The structure-oriented strategy derives tests for structural coverage criteria which are state coverage, transition coverage, and round-trip. The property-oriented strategy produces tests from the counterexamples of model checking. Strategies used are checking an aspect-oriented state model against trap properties and checking mutants of aspect models. The capabilities of the strategies in detecting faults are evaluated through mutation analysis of AOPs against the fault model. Results show that both strategies are complement to each other in detecting many aspect faults.

[72] D. Xu and J. Ding. Prioritizing state-based aspect tests. In *Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation*, pp. 265-274, 6-10 April, 2010.

Explores the prioritization of aspect tests, in incremental testing fashion, for aspect-oriented programs against their state model with transition coverage and round-trip coverage. Prioritization is done to report the failure earlier and to reduce test execution time. The prioritization is based on how aspects do the modifications base classes. The extent of modification which is the number of new and changed components in state transition identifies the priority of testing. Higher number of changes means higher priority for test generation. Test generation is done in Model-based Aspect Checking and Testing (MACT) framework [71]. Cases studies conducted to evaluate the effectiveness of the prioritization using finite state machines through mutation analysis of AspectJ programs show that prioritization has accelerated failure report.

[73] R. M. Parizi and A. A. A. Ghani. A theoretical evaluation of automated aspect-oriented programs testing approaches. In *Proceedings of the Annual International Conference on Software Engineering (SE 2010)*, pp. S11- S19, 2010.

Presents an overview and theoretical evaluation of approaches on automated AOP testing. The approaches surveyed are categorized as automated test generation and selection (Wrasp [14], Aspectra [22], Raspect [23],

APTE [24], and EAT [54]), automated test oracle (Pipa), and automated test execution (aUnit). The comparison is based on the following criteria; test type, test scope, automation level, and tool support.

[74] O. A. L. Lemos and P. C. Masiero. A poincut-based coverage analysis approach for aspect-oriented programs. Journal of Information Sciences, Vol. 181, pp. 2721-2746, 2011.

Presents a coverage analysis approach for exercising statements, branches, and def-use pairs of each advice at each affected join point. A control- and data-flow model, named as PointCut-based Def-Use Graph (PCDU), is proposed based on Java bytecode resulted from the compilation of AspectJ programs. From PCDU, two control-flow testing criteria (all-pointcut-based-advice-nodes and all-pointcut-based-advice-edges) and one data-flow testing criteria (all-pointcut-based-uses) are also proposed. The approach is implemented in JaBUTi/PC-AJ tool. Theoretical, empirical and exploratory studies were conducted and they show evidence that the approach is feasible and effective.

[75] A. Delamare, B. Baudry, S. Ghosh, S. Gupta, and Y. Le Traon. An approach for testing pointcut descriptors in AspectJ. Journal of Software Testing, Verification and Reliability, Vol. 21, pp. 215-239, 2011.

Presents an extension of their previous work in [57] and [58]. It focuses on the development of AdviceTracer [58], a tool to handle monitoring and storage of information related to the execution of advices, and AjMutator [57] to inject mutants by inserting faults in the PCDs. Detailed discussion on motivation and challenges for testing PCDs are also given. Empirical study conducted is of a larger scale as compared to their previous work [58]. The empirical study evaluates the effectiveness and utility of AdviceTracer with JUnit, in comparison with pure JUnit, for writing test cases in AspectJ PCDs. Results from the study shows that AdviceTracer produces simpler test cases (easier to write) and detect more faults than pure JUnit.

[76] R. M. Parizi and A. A. A. Ghani. On the preliminary adaptive random testing of aspect-oriented programs. In *Proceedings of the 6th International Conference on Software Engineering Advances (ICSEA 2011)*, Barcelona, Spain, pp. 49-57, 23-29 October, 2011.

Investigates the opportunities that adaptive random testing (ART) can provide to aspect-oriented programming. The investigation focuses on three perspectives; that are category and choice, object-based, and coverage-based. The perspectives are analyzed based on their underlying techniques and different distance measures, and their applicability to AOP. This is a preliminary investigation on the applicability of ART for AOP.

[77] R. M. Parizi, A. A. A. Ghani, R. Abdullah, and R. Atan. Empirical evaluation of the fault detection effectiveness and test effort efficiency of the automated AOP testing approaches. Journal of Information and Software Technology, Vol. 53, pp. 1062-1083, 2011.

Presents an empirical evaluation of the existing automated AOP testing approaches through experimentation. The automated AOP testing approaches concerned are Wrasp [14], Aspectra [22], Raspect [23], and EAT [54]. The focus of the evaluation is test input generation and selection strategies of the approaches with respect to effectiveness in detecting faults and required effort to detect the faults. The study makes use of mutation analysis on the four approaches through a process known as M-process. Adaptive AjMutator tool, an extension of AjMutator [57], is developed to generate mutants in the process. Results of the study reveal that EAT is more effective, but not significant for all approaches. In the case of test effort efficiency, Wrasp shows the lowest amount of test effort. Whereas EAT exhibits the highest amount of test effort. This means EAT is the most effective but with less efficient.

[78] P. Alves, A. Santos, E. Figueiredo, and F. Ferrari. How do programmers learn AOP? An exploratory study of recurring mistakes. In *Proceedings of 5th Latin-American Workshop on Aspect-Oriented Software Development (LA-WASP.11)*, Sao Paolo, Brazil, 26 September, 2011.

Reports the results of a series of experiments in characterizing mistakes made by novice and junior programmers. The results show that mistakes recur by programmers with specific background.

[79] F. C. Ferrari, A. Rashid, and J. C. Maldonado. Design of mutant operators for the AspectJ language. *Technical Report Version 1.0,* Computing Department, Federal University of Sao Carlos, Brazil, December 2011.

Provides the mutation operators for AspectJ language. Originally the operators are proposed in [46]. The operators are grouped into into pointcut descriptor (PCD), intertype declaration (ITD), and advices.

[80] N. McEachen and R. Alexander. Distributing classes with woven concerns – An exploration of potential fault scenarios. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development (AOSD'05)*, Chaicago, Illinois, pp. 192-200, 14-18 March 2005.

Explores the potential faults that could occur with respect to foreign aspects in AspectJ version 1.2. A foreign aspect is an aspect woven into a class or set of classes with the resultant bytecode being imported by another party not having access to the aspect code. The problem occurs because of unboubded pointcut and partial weaving.

[81] A. Restivo and A. Aguiar. Towards detecting and solving aspect conflicts and interferences using unit tests. *Workshop SPLAT'07*, Vancouver, British Columbia, Canada, 12-13 March 2007.

Presents a methodology that uses unit testing to detect conflicts and interferences introduced by aspects.

[82] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*

(ECOOP'97), Jyvaskyla, Finland, pp. 220-242, 9-13 June, 1997.

Reports on early development of aspect-oriented programming done at Xerox Palo Alto Research Center.

[83] G. J. Myers. The art of software testing, Wiley, New York, 1979.

Classical book on software testing and still has its influence today. A new version was revised and updated by Tom Badgett, Todd Thomas, and Corey Sandler. The version was published in 2004. The updated version contains examples of current programming languages, testing for extreme programming and e-commerce.

[84] B. Beizer. Software Testing Techniques. Second edition. 1990

A very comprehensive book on the testing techniques. Many testing techniques are enumerated and discussed in detail such as domain testing, data-flow testing, transaction-flow testing, syntax testing, logic-based testing, etc.

[85] W. Hetzel and B. Hetzel. The complete guide to software testing. 2nd edition, John Wiley and Son, Inc. New York, 1991.

Presents a new perspective on software testing as a life cycle activity. It covers the concepts and principles of testing, offering detailed discussions of testing techniques, methodologies and management viewpoints. This is a revised edition of its 1988's edition featuring new chapters on testing methodologies such as ANSI standard-based testing and a survey of testing practices.

[86] P. Amman and J. Offutt. Introduction to Software Engineering. Cambridge University Press, New York, 2008.

Presents techniques related to dynamic or execution-based testing. The approach taken in organizing the presentation of the techniques is software testing coverage criteria since software testing is based on satisfying coverage criteria. The coverage criteria concerned are based on graphs, logical expressions, input space, and syntax structures.

[87] R.Laddad. AspectJ in Action: A Practical Aspect-Oriented Programming. Manning Publications Co. New York. 2003.

The book explains the AOP methodology and AspectJ language. It also presents examples of how AspectJ is used as solution to common concerns such as logging, policy enforcement, resource pooling, business rules, thread-safety, authentication and authorization, as well as transaction management. Its new edition was published in 2009.

[88] N. Juristo, A. M. Moreno, and S. Vegas. Reviewing 25 years of testing technique experiments. Journal of Empirical Software Engineering, Vol. 9, pp. 7-44, 2004.

Analyzes the maturity level of the knowledge about testing techniques.

[89] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs, in: ACM SIGSOFT Foundation of Software Engineering, Newport Beach/CA, ACM Press, USA, pp. 147-158, 2004.

In the perspective of this annotated bibliography, Stack program inside this paper has been used as one of the subjects for experimentation conducted by researchers.

Abdul Azim Abdul Ghani received the B.Sc. in Mathematics/Computer Science from Indiana State University, USA in 1984 and M.Sc. in Computer Science from University of Miami, USA in 1985. He joined University Putra Malaysia in 1985 as a lecturer in Computer Science. He received the Ph.D in Software Engineering from University of Strathclyde in 1993. He is a Professor at the Department of Information System, University Putra Malaysia. His research interests are Software engineering, Software measurement, Software testing, and Aspect-oriented programming (AOP).

Reza Meimandi Parizi is a researcher in the Department of Software Engineering at University of Malaya. His research interests in software engineering include automated software testing, aspect-oriented programming, software traceability, and empirical studies. He holds a Ph.D degree in Software Engineering from the Universiti Putra Malaysia.