Journal of Software Engineering and Applications, **2013, 6, 5-13**
http://dx.doi.org/10.4236/jsea.2013.610A002 Published Online October 2013 (http://www.scirp.org/journal/jsea)

Towards Semantic Mutation Testing of Aspect-Oriented Programs

Abdul Azim Abdul Ghani

Department of Software Engineering and Information System, Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, Serdang, Malaysia.
Email: azim@upm.edu.my

Received July 30th, 2013; revised August 28th, 2013; accepted September 5th, 2013

ABSTRACT

Aspect-oriented programs have received much attention from software testing researchers. Various testing techniques and approaches have been proposed to tackle issues and challenges when testing aspect-oriented programs including traditional mutation testing. In traditional mutation testing of aspect-oriented programs, mutants are generated by making small changes to the syntax of the aspect-oriented language. Recently, a new approach known as semantic mutation testing has been proposed. This approach mutates the semantics of the language in which the program is written. The mutants generated misunderstandings of the language which are different classes of faults. Aspect-oriented programming presents itself with different properties that can be further explored with respect to semantic mutation testing. This paper describes various possible scenarios that semantic mutation testing strategy might have particular value in testing aspect-oriented programs.

Keywords: Aspect-Oriented Program Testing; Mutation Testing; Semantic Mutation Testing

1. Introduction

Aspect-Oriented Programming (AOP) [1,2] was first introduced in the middle of 1990s at the Xerox Palo Alto Research Center. AOP provides means for modularizing and separating crosscutting concerns in which it produces a system with higher degree of modularity than the other paradigms such as Object-Oriented Programs (OOPs). However, since it has new constructs and properties that other programming paradigms do not have, it brings new challenges and aspect-related defects/faults [3,4], which are not present when testing other types of programs, which in turn cannot be addressed using traditional unit or integration testing approaches [5,6]. For instance, an aspect is given in a bank system that is supposed to implement authentication (as crosscutting concern) before calling to a set of methods. AOP can get this requirement done by simply capturing any call to the given methods (*i.e.* join points) in the core code by means of pointcuts and then injects those identification functionalities or behaviors (*i.e.* advice) before the method which were invoked by callers. More specifically, if AOP misses to capture some calls to the given methods therefore the authentication will not properly be applied and it may cause a severe failure in the system.

Therefore, this programming paradigm, although enhances the modularity, cannot provide correctness by itself and thus like any other programs, it is prone to errors (by developer, programmer, etc.) and requires the use of software testing strategy to produce validated and high quality AO software. Software testing is an essentially valuable practice to ensure correctness of a program in finding defects.

Testing is a central issue in aspect-oriented software development. Since aspect-oriented programming introduces new constructs and programming means for separation of concerns, testing of AO programs is more sophisticated and challenging in which the existing techniques for testing cannot accommodate this matter and need to be leveraged or extended. Furthermore, research in AO testing has focused on approaches such as code-based structural testing [6,7], specification-based functional testing [8-10], use of random testing [11], and mutation testing [12]. Surveys by [13] and [14] have examined the effectiveness of testing techniques for AO programs which include data flow based unit testing, state-based approach, aspect flow graph based technique, unit testing aspectual behaviour, and model-based approach. Reference [15] provides an

annotated bibliography of aspect-oriented program testing.

In the context of this paper, mutation testing [16] (called also as traditional mutation testing) is the basis. Mutation testing aims to produce test cases that are good at distinguishing some description and variants of them. This is done by producing mutants as a result of applying mutation operators. In traditional mutation testing, the mutation operators work at the syntactic level. Thus, the mutants do not consider any mistakes or errors due to semantic misunderstanding. A new strategy known as semantic mutation testing [17,18] has shown that it is promising in tackling certain aspect of software defects. However this strategy has not yet been investigated for AO testing. Therefore, the purpose of this paper is to explore a semantic mutation testing strategy for testing aspect-oriented programs.

The rest of the paper is structured as follows: Section 2 describes traditional mutation testing. Section 3 discusses mutation testing applied to aspect-oriented programming. Section 4 explains about semantic mutation testing. Section 5 discusses the issues and challenges faced in testing aspect-oriented software. Section 6 provides ways forward in dealing with semantic mutation testing for aspect-oriented software. These are scenarios that may have particular value to be researched. Section 7 is the conclusion.

2. Mutation Testing (Traditional)

Mutation testing (also known as mutation analysis [16]) is an approach originally to automate testing with the purpose to produce test cases that are effective at distinguishing between a computer program and its variants. Over the years, mutation testing has been remarkably studied and applied not only to programs from different programming paradigms such as Fortran programs [19, 20], C programs [21], Java programs [22], and AspectJ programs [12,23], but also to specifications or models of programs, such as Finite State Machine [24], Petri Nets [25], and Security Policies [26]. Surveys and reviews on work in mutation testing can be found in [27-29].

The main idea behind this approach is producing mutants by introducing changes to the program. These changes imitate classes of faults and test cases produced are used to distinguish the original program from its mutants. The process begins by systematically seeding faults into a program. These faults are introduced through application of some kind of predefined operators called *mutation operators* to the original code. The mutation operators involve syntactic changes. For example, replacing a variable with a constant, replacing + with −, or replacing > with >=. The mutation operators themselves are typically derived from fault models for the specific programs in context.

Each fault introduced results in a faulty version of program called a *mutant* that slightly differs from the original program. Then, the purpose is to run test cases generated on the mutants to see if any of the test cases distinguishes the faults introduced earlier into the code. If a mutant produces different outputs than the original program then the mutant is *killed* by the test cases. Otherwise, the mutant is classified as either *equivalent* or *live* mutant. Moreover, mutants that are functionally equivalent, although syntactically different to the original program, always produce the same output and thus no test cases will be able to kill them. The mutants are called *undetected* or *live* mutants.

In traditional mutation testing, mutation operators are introduced at syntactic level. This represents errors due to small slips or typos. Mutants that are produced only represent syntactically different programs, but do not represent misunderstanding of semantic mistake. Thus, to complement the traditional mutation testing, semantic mutation testing was proposed. Section 4 introduces semantic mutation testing.

3. Mutation Testing for Aspect-Oriented Programs

The introduction 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, which could 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.

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.

AspectJ [30], an extension of Java language, is widely studied aspect-oriented programming language. 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 providing many special constructs such as aspects, advice, joinpoints, and pointcuts.

As far as mutation testing for aspect-oriented programs is concerned, the technique focuses on syntactic constructs of aspect-oriented programs to model faults by means of mutation operators for aspect-oriented languages. References [4,31,32] generalize faults for general aspect-oriented programs and produced three groups of mutation operators. The mutation operators are grouped into *pointcut descriptor* (PCD), *intertype declaration* (ITD), and *advices*. **Table 1** shows examples of aspect-oriented fault types listed according to each group. These groups were obtained from a thorough analysis of various work on aspect-oriented fault models and types [3,33-42], fault classification [43], and bug patterns [44]. Other work that focus on mutation operators for aspect-oriented programs are [45-47]. **Table 2** lists mutation operators for aspect-oriented programs at pointcut and advice levels for AspectJ language.

Table 1. Examples of fault types for AOP.

Description
Pointcut related faults
Selection of a superset of join point
Selection of a subset of join point
Selection of a wrong set of join points, including intended and unintended ones
Incorrect use of primitive pointcut designators
Incorrect pointcut composition rules
Incorrect matching based on dynamic values and events
Inter-type declaration related faults
Improper method introduction, resulting in inconsistencies in method overriding
Incorrect changes in class hierarchy
Incorrect or omitted aspect precedence declaration
Advice related faults
Incorrect advice type specification
Incorrect control or data flow due to execution of the original join point
Incorrect access to join point static information
Advice bound to incorrect pointcut

Table 2. Examples of mutation operators for AOP.

Mutation operators	Description
	Pointcut level
PWIW	Inserts wildcards into pointcut expressions
PWAR	Removes annotation tags from type, field, method and constructor patterns
PSWR	Removes wildcards from pointcut expressions
POPL	Changes the parameter lists of primitive Pointcut Designators/Descriptors (PCDs)
POEC	Adds, omits or alters exception throwing clauses
PCTT	Replaces a this PCD with a target one and vice versa
PCCE	Replaces a call PCD with an execution/initialization/ preinitialization PCD and vice versa
PCGS	Replaces a get PCD with a set one and vice versa
PCLO	Changes the logical operators in PCDs compositions
PCCC	Replaces a cflow PCD with a cflowbelow one and vice versa
	Advice level
ABAR	Replaces a before clause with an after (returning/throwing) one and vice versa
APSR	Removes invocations to proceed statement
APER	Removes guard conditions which surround proceed statements
AJSC	Replaces a thisEnclosingJoinPointStaticPart reference with a thisJoinPointStaticPart one and vice versa
ABHA	Removes implemented advice
ABPR	Changes pointcut-advice binding by replacing pointcuts which are bound to advice

Most work on mutation testing of aspect-oriented programming focus AspectJ programs. Reference [48] proposes a framework to automatically generate mutants for pointcuts and to detect equivalent mutants. A tool that implements the framework for mutation testing on pointcut expression was proposed. A tool known as AjMutator [49] was proposed to generate and detect mutants related to pointcut descriptors. The tool detects equivalent mutants by leveraging on the static analysis of the compiler. Other tools that were proposed to automate generation of mutants for AspectJ programs are MuAspectJ [50] and Proteum/AJ [51].

4. Semantic Mutation Testing

Semantic mutation testing concept was proposed by [17,18]. The idea behind this concept is that a description language with semantics is mutated to represent likely misunderstandings. Thus, it allows us to explore possible variation of the semantics. In the case of programming languages, there are situations that need specific imple-

mentation that lead to different version of compilers being written, for example in handling of precision of floating point [52]. As a consequence, a program involving manipulation of floating point numbers will semantically wrong in certain compilers. Mutation operators could be applied to the semantics of the language to reflect the different or alternative semantics.

Semantic mutation testing formally can be represented as follows [18]: Given a source code N written in a programming language with semantic L, the behavior is defined by the pair (N, L). The application of a semantic mutation operator is of the form $(N, L) \rightarrow (N, L')$ will produce a first-order mutant (N, L') of (N, L) by applying one mutation operator once to the semantics of the language. Suppose that (N, L') is mutated from (N, L). Then N has its meaning under L represented by N_L and its meaning under L' represented by $N_{L'}$. Given a test case t, $N_L(t)$ is the behavior produced when applying t to N under semantics L and $N_L(t)$ is the behavior produced when applying t to N under semantics L'. The mutant (N, L') is said to be *killed* by a test case t if and only if $N_L(t) \neq N_L(t)$. Moreover, this mutant (N, L') is an equivalent mutant if $\forall_t | N_L(t) = N_L'(t)$.

In mutation testing of a source code N with semantics L, a set of mutation operators are applied individually to L. This will produce alternative semantics $L_1,\ldots,L_m$. The mutants $(N, L_1),\ldots,(N, L_m)$ are used to evaluate a test suite or to drive test generation in which a test suite should kill every non-equivalent mutant in the set of mutants.

According to [17,18], three approaches to implement semantic testing are:

- Have a parameterisable system for interpreting a language, the parameters allowing the semantic to be mutated.
- Express the semantics in the form that can be manipulated.
- Simulate a mutation of the semantic by making changes to the syntax of the description.

Besides the work in [52], semantic mutation testing for C language has also being implemented by using the third approach in which semantic mutation are simulated by making changes to the syntax of C program [53]. A tool known as SMT-C was implemented to handle this. In this tool, they implement a group of 13 semantic mutation operators based on specific misunderstandings of C languages including 4 related to floating point numbers.

As far as we are concerned from our reading, there is no work on semantic mutation testing for aspect-oriented program. Only a few works using traditional mutation testing have been reported in the literature as mentioned in the above section. In traditional mutation testing, the mutation operators work at the syntactic level and mostly focusing on AspectJ language. Thus, the mutants produced do not consider any mistakes or errors due to semantic misunderstanding of the aspect-oriented programming language. Semantic mutation testing has shown that it is promising in tackling certain type of software defects, for example, floating-point numbers. However this strategy has not been yet applied to AOP testing.

5. Issues and Challenges of AO Programs

Aspect-oriented programming paradigm new concepts and properties extend the capabilities of other programming paradigms. In AOP, separate aspects are defined (or coded) to contain crosscutting actions (obliviousness nature). These aspects then are woven into classes to represent the core concerns of the system. The woven code may vary between different compilers and versions. Such concepts and properties pose new challenges and issues regarding testing. Testing aspect-oriented programs must consider faults due to an aspect code fragments or where the fragments are inserted through pointcuts. Furthermore the types of code inserted by an aspect can be different, thus different aspects may need different testing strategies. Up to now many testing strategies has been proposed.

The proposed testing strategies are the outcomes from trying to handle the issues [3,54-56] 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. Even though the current testing strategies test AOP statically, we believe that the semantics of AOP influence the behavior of the program.

Many researchers have investigated techniques with the goal of understanding and verifying the effects of aspect on the core concern as well as an aspect interaction as the result of obliviousness nature of AOP. References [57,58] pointed out that new aspects introduce into a program may cause certain modules cease to function as expected due to conflicts between superimposed aspects. Thus the occurrence of the problems may be unpredictable. This problem (also called semantic interference conflicts) can occur when different aspects possibly developed by different developers at different time are superimposed on the same join point may semantically interfere with each other. Potential conflicts may occur depending on the model used or specification style for expressing aspects [57].

These drawbacks are reflected as composition problems between aspects and base code. An aspect composition problem refers to an incorrect behaviour in an application that is related to the use of AOP language. Work to determine this category of problems has been reported in recent years [59-62]. For example [61] classifies aspect interference as the following:

- Wildcards Pointcut Problem that refers to the use of wildcards characters can lead to accidental joinpoints capture and miss.
- Conflicts between Aspects that refers to the order of weaving of a set of aspects is very important to ensure correct behaviour.
- Circular Dependency between Aspects that refers to the formation of circles of the semantic dependencies between aspects.
- Conflicts between Concerns that refers to functionality needed by a concern is changed by another concern.

A more recent work on the classification of aspect composition problems is the work of [63]. The classification is a three-dimensional taxonomy of aspect composition problems which include the following categories:

- Functional versus Crosscutting Aspect Composition Problems classify problems that impact the functionalities of an aspect or a base program, and problems that prevent the correct implementation of crosscutting concerns respectively.
- Inter-Aspects Problems versus System-Wide Problems refers respectively to the aspect compositions problems during aspectual composition process, and problems involving both aspect and base programs.
- Semantic versus Syntax Problems refers problems that occur due to inconsistencies that change the meanings or logic of an AO programs, and problems that are incurred by the insufficient of AOP languages to support implementation of crosscutting concerns.

The issues raised by the researchers are not only concerned with syntax of AO program, but also the semantics of it.

6. Way Forward

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 behaviour on aspects are merged to the core concern code to yield a working system. However as being mentioned in previous section not all types of fault can be detected by testing techniques. This situation provides avenue for more research in AO testing as up to recently researchers are focusing on testing syntactic-oriented of AO programs. Thus, inspired by semantic mutation testing technique, it is also desirable to study into detail the applicability of the technique. Below are some possible scenarios that can be further explored.

- Model transformation–refinement and translation

In aspect-oriented software development, various description of the underlying aspect-oriented software may be generated by different activities. Requirements and designs artifacts are transformed to aspect-oriented language such as AspectJ or AspectC++. UML diagrams have been extended for aspect-oriented modeling [64] at the requirement and design levels. A more recent notation is aspect-oriented user requirements notation [65,66] to depict user requirements with respect to aspects has been proposed which later can be transformed to aspect-oriented design model [67]. The form of description of aspect-oriented software changes from abstract models to concrete code in aspect-oriented programming language. As transformation moves to the target description, semantic misunderstanding can be introduced because of the informality of either description or the semantic differences between the source and target description. Semantic mutation operators could change the semantics of the target description to simulate the semantics given to the source description.

- Common misunderstanding

For a given set common misunderstandings for a par-

ticular aspect-oriented description, a set of semantic mutation operators could represent these misunderstandings. Such a set of operators could be formed by analysis and studies of common faults; such as the studies on mutation analysis for model transformation [68], on mutation operators for AspectJ language [31], on fault model for the aspect composition at design level [41], and on the recurring mistakes by programmers [69]. Given a mutation operator representing a misunderstanding, a test set that kills the mutants generated by this operator should be good at locating faults that due to the misunderstanding.

- Refactoring

Misunderstandings may occur through refactoring. Refactoring is a process of changing a software system in such a way to improve the code's internal structure without changing its external behaviour. In the perspective of aspect-oriented applications, refactoring can be applied in situation where an aspect can be introduced to an existing object-oriented program or legacy system, for example, refactoring a Java program to an equivalent AspectJ program. The legacy system and the new system may exhibit different semantics. However, support for assuring that refactoring preserves behaviour is lacking. Thus semantic mutation testing seems logical to be applied in order to test semantic differences between the two systems. References [35,70] are worth to be considered involving refactoring of aspects and testing.

- Migration

It can happen that a specific aspect-oriented language used needs to be migrated to a different programming language. For example migrating from AspectJ to AspectC++. The original language and the new one may encompass different semantics. There is a tendency that mistakes will be caused by this difference in semantics. Further analysis of the languages need to be done to determine the possible different in semantics. The process of migration could be assisted using a tool to generate test case that can locate mistakes caused by the differences in semantics. Mutation operators defined depend on the original and new language semantics.

- Aspect Composition

Multiple pieces of advice can be applied to the same join point. Advice precedence determines the order in which advice is woven. In AspectJ programming language, precedence is dealt with differently depending on where the pieces of advice are defined. It can be either in the same aspect or in different aspects. AspectJ programmers have the option of declaring the order in which aspects are woven by a precedence statement. It can happen that no precedence statement is declared. Thus, the precedence of aspects is undefined in the semantics of AspectJ. In this case, the AspectJ compiler chooses an order in which to weave aspects. Generally this order cannot be inferred by programmers prior to weaving. Unfortunately, different weaving orders can result in programs that behave differently. We need to know the weaving order to be able to predict the result. If the weaving order is undetermined, as in the absence of a precedence declaration, the woven program will at the least be not portable (since different compilers can choose different weaving orders). Moreover, programmers will not be informed about the order the compiler chooses. In another perspective, different pieces of advice can appear within aspects in a certain textual order that their precedence follows certain predetermined rules. However these rules lead to the problems of circularity and unable to express all weaving orders. In such cases, programmers must manually modify the order the advice is listed in the program text to ensure the resulting weaving order eliminates circularity, and produce a semantically appropriate weaving for the task at hand which is a non-trivial and lengthy process. However, if the programmers forget to do that or if it is not obvious to find the precedence relation due to a big number of aspects or the complexity of their properties then the result can be disastrous. Semantic mutation testing can be used to explore the impact of undeclared precedence of pieces of advice and aspects, and thus assists in determining of portability of the program. Semantic mutation operator would rearrange the order of the precedence. If the mutation operator generates an equivalent mutant then the behaviour of the program of the tested program is not affected by portability issues. The above scenario on aspect composition only involves precedence. There are other problems that can occur in aspect composition as mentioned in [59-63]. These problems provide opportunity for applying semantic mutation testing.

7. Conclusion

In this paper, the new type of mutation testing called semantic mutation testing which has been introduced is advocated to be applied in testing aspect-oriented program. The aim is to represent mistakes due to the misunderstandings of semantics of aspect-oriented programs. A range of scenarios in which semantics mutation testing may have specific value has been initially described. The scenarios provide opportunities for further research in this area.

8. Acknowledgements

The author would like to thank Malaysia Ministry of Education for providing grant under Fundamental Research Grant Scheme (08-01-13-1222FR) for the project.

REFERENCES

[1] G. A. Colyer and A. Clement, "Aspect-Oriented Programming with AspectJ," *IBM Systems Journal*, Vol. 44, No. 2, 2005, pp. 301-308.

http://dx.doi.org/10.1147/sj.442.0301

[2] G. Kiczales, J. Lamping, C. V. Lopes, J. J. Hugunin, E. A. Hilsdale and C. Boyapati, "Aspect-Oriented Programming," US Patent No. 6467086, 2002.

[3] R. T. Alexander, J. M. Bieman and A. A. Andrews, "Towards the Systematic Testing of Aspect-Oriented Programs," Technical Report, Colorado State University, Fort Collins, 2004.

[4] F. C. Ferrari, J. C. Maldonado and A. Rashid, "Mutation Testing for Aspect-Oriented Programs," *Proceedings of the* 1*st International Conference on Software Testing*, *Verification*, *and Validation*, Norway, 9-11 April 2008, pp. 52-61.

[5] M. Harman, F. Islam, T. Xie and S. Wrappler, "Automated Test Data Generation for Aspect-Oriented Programs," *Proceedings of the* 8*th International Conference on Aspect-Oriented Software Development*, Charlottesville, 2-6 March 2009, pp. 185-196.

[6] J. Zhao, "Dataflow-Based Unit Testing of Aspect-Oriented Programs," 27*th Annual IEEE International Computer Software and Applications Conference* (*COMPSAC*'2003), Dallas, 3-6 November 2003, pp. 188-197.

[7] 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," *Journal of Systems and Software*, Vol. 80, No. 6, 2007, pp. 862-882. http://dx.doi.org/10.1016/j.jss.2006.08.022

[8] 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, 2008, pp. 413-437. http://dx.doi.org/10.1142/S0218194008003672

[9] D. Xu and W. Xu, "State-Based Incremental Testing of Aspect-Oriented Programs," *Proceedings of the* 5*th International Conference on Aspect Oriented Software Development* (*AOSD*'06), Bonn, 20-24 March 2006, pp. 180-189.

[10] W. Xu and D. Xu, "State-Based Testing of Integration Aspects," *Workshop on Testing Aspect-Oriented Programs* (*WTAOP*'06), Portland, 17-20 July 2006, pp. 7-14.

[11] R. M. Parizi, A. A. Abdul Ghani, R. Abdullah and R. Atan, "On the Applicability of Random Testing of Aspect-Oriented Programs," *International Journal of Software Engineering and Its Application*, Vol. 3, No. 3, 2009, pp. 1-19.

[12] F. C. Ferrari, A. Rashid and J. C. Maldonado, "Towards the Practical Mutation Testing of AspectJ Programs," *Science of Computer Programming*, Vol. 78, No. 9, 2013, pp. 1639-1662. http://dx.doi.org/10.1016/j.scico.2013.02.011

[13] S. A. Ali Naqvi, S. Ali and M. Uzair Khan, "An Evaluation of Aspect-Oriented Testing Techniques," *Proceedings of the* 2005 *International Conference on Emerging Technologies*, Islamabad, 17-18 September 2005, pp. 461-466.

[14] R. M. Parizi and A. A. Abdul Ghani, "A Survey on Aspect-Oriented Testing Approaches," *Proceedings of the* 5*th International Conference on Computational Science and Applications*, Malaysia, 26-29 August 2007, pp. 78-85.

[15] A. A. Abdul Ghani and R. M. Parizi, "Aspect-Oriented Program Testing: An Annotated Bibliography," *Journal of Software*, Vol. 8, No. 6, 2013, pp. 1281-1300.

[16] R. A. DeMillo, R. J. Lipton and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *IEEE Computer*, Vol. 11, No, 4, 1978, pp. 34-41. http://dx.doi.org/10.1109/C-M.1978.218136

[17] J. A. Clark, H. Dan and R. M. Hieron, "Semantic Mutation Testing," *Proceedings of the* 3*rd International Conference on Software Testing*, *Verification*, *and Validation Workshops*, Paris, 6-10 April 2010, pp. 100-109. http://dx.doi.org/10.1109/ICSTW.2010.8

[18] J. A. Clark, H. Dan and R. M. Hieron, "Semantic Mutation Testing," *Science of Computer Programming*, Vol. 78, No. 4, 2013, pp. 345-363. http://dx.doi.org/10.1016/j.scico.2011.03.011

[19] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton and F. G. Sayward, "Mutation Analysis," Technical Report GIT-ICS-79/08, Georgia Institute of Technology, Atlanta, 1979.

[20] K. N. King and J. Offutt, "A Fortran Language System for Mutation-Based Software Testing," *Software*: *Practice and Experience*, Vol. 21, No. 7, 1991, pp. 685-718. http://dx.doi.org/10.1002/spe.4380210704

[21] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur and E. Spafford, "Design of Mutant Operators for the C Programming Language," Technical Report SERC-TR-41-P, Purdue University, Lafayette, 1989.

[22] P. Chevalley and P. Thevenod-Fosse, "A Mutation Analysis Tool for Java Programs," *International Journal of Software Tools for Technology Transfer*, Vol. 5, No. 1, 2002, pp. 90-103. http://dx.doi.org/10.1007/s10009-002-0099-9

[23] P. Anbalagan and T. Xie, "Efficient Mutant Generation Testing of Pointcuts in Aspect-Oriented Programs," *Proceedings of the* 2*nd workshop on Mutation Analysis*, 2006, pp. 51-56.

[24] S. P. F. Fabbri, M. E. Delamaro, J. C. Maldonado and P. Masiero, "Mutation Analysis Testing for Finite State Machine," *Proceedings of the* 5*th International Symposium on Software Reliability Engineering*, Monterey, 6-9 November 1994, pp. 220-229.

[25] S. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro and W. E. Wong, "Mutation Testing Applied to Validate Specification Based on Petri Nets," *Proceedings of IFIP TC*6 8*th International Conference on Formal Description Techniques VIII*, Vol. 43, 1995, pp. 329-337.

[26] E. E. Martin and T. Xie, "A Fault Model and Mutation Testing of Access Control Policies," *Proceedings of the* 16*th International Conference on World Wide Web*, May 2007, pp. 667-676. http://dx.doi.org/10.1145/1242572.1242663

[27] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Transactions on Software Engineering*, Vol. 37, No. 5, 2011, pp. 649-

678. http://dx.doi.org/10.1109/TSE.2010.62

[28] J. Offut, "A Mutation Carol: Past, Present and Future," *Information and Software Technology*, Vol. 53, No. 10, 2011, pp. 1098-1107. http://dx.doi.org/10.1016/j.infsof.2011.03.007

[29] J. Offut and R. H. Untch, "Mutation 2000: Uniting the Orthogonal," *Proceedings of Mutation* 2000: *Mutation Testing in the Twentieth and Twenty First Centuries*, San Jose, October 2000, pp. 45-55.

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

[31] F. C. Ferrari, A. Rashid and J. C. Maldonado, "Design of Mutant Operators for the AspectJ Language," Technical Report Version 1.0, University of Sao Carlos, Sao Carlos, 2011.

[32] 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," *Proceedings of the* 2010 *Brazilian Symposium on Software Engineering*, Salvador, 27 September-1 October, 2010, pp. 50-59.

[33] M. Ceccato, P. Tonella and F. Ricca, "Is AOP Code Easier or Harder to Test than OOP Code?" *Proceedings of the Workshop on Testing Aspect-Oriented Programs*, Chicago, March 2005.

[34] N. McEachen and R. T. Alexander, "Distributing Classes with Woven Concerns—An Exploration of Potential Fault Scenarios," *Proceedings of the* 4*th International Conference on Aspect-oriented Software Development* (*AOSD*' 05), Chicago, 14-18 March, 2005, pp. 192-200. http://dx.doi.org/10.1145/1052898.1052915

[35] 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], 2005.

[36] J. S. Baekken and R. T. Alexander, "Towards a Fault Model for AspectJ Programs: Step 1—Pointcut Faults," *Proceedings of the* 2*nd Workshop on Testing Aspect-Oriented Programs*, Portland, 20 July 2006, pp. 1-6. http://dx.doi.org/10.1145/1146374.1146375

[37] J. S. Baekken and R. T. Alexander, "A Candidate Fault Model for AspectJ Pointcuts," *Proceedings of the* 17*th International Symposium on Software Reliability Engineering* (*ISSRE*'06), Raleigh, 7-10 November 2006, pp. 169-178.

[38] J. S. Baekken, "A Fault Model for Pointcuts and Advice in AspectJ Programs," Master Thesis, School of Electrical Engineering and Computer Science, Washington State University, Pullman, 2006.

[39] C. Zhao and R. Alexander, "Testing AspectJ Programs Using Fault-Based Testing," *Proceedings of the* 3*rd Workshop on Testing Aspect-Oriented Programs*, Vancouver, 12-13 March 2007, pp. 13-16.

[40] 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," *Proceedings of the* 6*th International Conference on the Quality of Information and Communication Technology*, Lisbon, 12-14 September 2007, pp. 65-76.

[41] 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, 2009, pp. 1-6. http://dx.doi.org/10.1145/1457516.1457521

[42] N. Kumar, A. Rathi, D. Sosale and S. N. Konuganti, "Enabling the Adoption of Aspects—Testing Aspects: A Risk Model, Fault Model and Patterns," *Proceedings of the* 8*th International Conference on Aspect-Oriented Software Development* (*AOSD*'09), Charlottesville, 2-6 March 2009, pp. 197-206.

[43] O. A. Lemos, F. C. Ferrari, P. C. Masiero and C. V. Lopes, "Testing Aspect-Oriented Programming Pointcut Descriptors," *Proceedings of the* 2*nd Workshop on Testing Aspect-Oriented Programs*, New York, 20 July 2006, pp. 33-38. http://dx.doi.org/10.1145/1146374.1146380

[44] S. Zhang and J. Zhao, "On Identifying Bug Patterns in Aspect-Oriented Programs," *Proceedings of the* 31*st Annual International Computer Software and Applications Conference* (*COMPSAC* 2007), Beijing, 24-27 July 2007, pp. 431-438. http://dx.doi.org/10.1109/COMPSAC.2007.159

[45] 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, 2004.

[46] M. Mortensen, and R. T. Alexander, "An Approach for Adequate Testing of AspectJ Programs," *Proceedings of the* 1*st Workshop on Testing Aspect-Oriented Programs —In Conjunction with AOSD*'2005, Chicago, 14-18 March 2005.

[47] M. Singh and S. Mishra, "Mutant Generation for Aspect-Oriented Programs," *Indian Journal of Computer Science and Engineering*, Vol. 1, No. 4, 2010, pp. 409-415.

[48] P. Anbalagan and T. Xie, "Automated Generation of Pointcut Mutants for Testing Pointcuts in AspectJ Programs," *Proceedings of the* 19*th International Symposium on Software Reliability Engineering*, Seattle, 10-14 November 2008, pp. 239-248.

[49] R. Delamare, B. Baudry and Y. Le Traon, "AjMutator: A Tool for the Mutation Analysis of AspectJ Pointcut Descriptors," *Proceedings of the IEEE International Conference on Software Testing*, *Verification and Validation Workshops* (*ICSTW* 2009), Denver, 1-4 April 2009, pp. 200-204. http://dx.doi.org/10.1109/ICSTW.2009.41

[50] A. Jackson and S. Clarke, "MuAspectJ: Mutant Generation to Support Measuring the Testability of AspectJ Programs," Technical Report (TCD-CS-2009-38), ACM, 2009.

[51] F. C. Ferrari, A. Rashid, E. Y. Nakagawa and J. C. Maldonado, "Automating the Mutation Testing of Aspect-Oriented Java Programs," *Proceedings of the* 5*th Workshop on Automation of Software Test* (*AST*'10), Cape Town, 3-4 May 2010, pp. 51-58. http://dx.doi.org/10.1145/1808266.1808274

[52] H. Dan and R. M. Hierons, "Semantic Mutation Analysis of Floating-Point Comparison," *Proceedings of IEEE* 5*th*

International Conference on Software Testing, Verification and Validation, Montreal, 17-21 April 2012, pp. 290-299.

[53] H. Dan and R. M. Hierons, "SMT-C: A Semantic Mutation Testing Tool for C," *Proceedings of IEEE 5th International Conference on Software Testing, Verification and Validation*, Montreal, 17-21 April 2012, pp. 654-663.

[54] 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, 2008.

[55] R. M. L. M. Moreira, A. C. R. Paiva and A. Aguiar, "Testing Aspect-Oriented Programs," *Proceedings of the 5th Iberian Conference on Information Systems and Technologies*, Santiago de Compostela, 16-19 June 2010, pp. 1-6.

[56] A. Restivo and A. Aguiar, "Towards Detecting and Solving Aspect Conflicts and Interferences Using Unit Tests," *Workshop SPLAT*'07, Vancouver, 12-13 March 2007.

[57] L. Bergmans, "Towards Detection of Semantic Conflicts between Crosscutting Concerns," *AAOS* 2003 (Analysis of Aspect-Oriented Software), Darmstadt, 21 July 2003.

[58] P. Durr, T. Staijen, L. Bergmans and M. Aksit, "Reasoning about Semantic Conflicts between Aspects," *European Interactive Workshop on Aspects in Software*, EIWAS, Brussels, 1-2 September 2005.

[59] L. Bussard, L. Carver, E. Ernst, M. Jung, M. Robillard and A. Speck, "Safe Aspect Composition," *Workshop on Aspects and Dimensions of Concern at ECOOP*'2000, Cannes, June 2000.

[60] W. Havinga, I. Nagy and L. Bergmans, "An Analysis of Aspect Composition Problems," *Third European Workshop on Aspects in Software* 2006, Enschede, 31 August 2006, pp. 1-8.

[61] F. Tessier, M. Badri and L. Badri, "A Model-Based Detection of Conflicts between Crosscutting Concerns: Towards a Formal Approach," *International Workshop on Aspect-Oriented Software Development*, China, September 2004.

[62] H. Chengwan, L. Zheng and H. Keqing, "Towards Trusted Aspect Composition," *Proceedings of IEEE 8th International Conference on Computer and Information Technology Workshops*, Sydney, 8-11 July 2008, pp. 643- 648.

[63] K. Tian, K. Cooper, K. Zhang and H. Yu, "A Classification of Aspect Composition Problems," *Proceedings of the Third IEEE International Conference on Secure Software Integration and Reliability Improvement*, Shanghai, 8-10 July 2009, pp. 101-109. http://dx.doi.org/10.1109/SSIRI.2009.33

[64] W. Xu and D. Xu, "A Model-Based Approach to Test Generation for Aspect-Oriented Programs," *Workshop on Testing Aspect-Oriented Programs* (*AOSD*2005), Chicago, 14-18 March 2005.

[65] G. Mussbacher, "Aspect-Oriented User Requirements Notation," Ph.D. Thesis, SITE, University of Ottawa, Canada, 2010.

[66] G. Mussbacher, D. Amyot, J. Araujo and A. Moreira, "Requirements Modeling with the Aspect-Oriented User Requirements Notation (AoURN): A Case Study," In: S. Katz, M. Mezini and J. Kienzle, Eds., *Transactions on Aspect-Oriented Software Development* VII, Springer, Berlin, 2010, pp. 23-68.

[67] S. Mosser, G. Mussbacher, M. Blay-Fornarino and D. Amyot, "From Aspect-Oriented Requirements Models to Aspect-Oriented Business Process Design Models—An Iterative and Concern-Driven Approach for Software Engineering," *Proceedings of 10th International Conference on Aspect-Oriented Software Development* (*AOSD* 2011), Porto de Galinhas, 21-25 March 2011, pp. 31-42 .

[68] J. M. Mottu, B. Baudry and Y. Le Traon, "Mutation Analysis Testing for Model Transformations," In: A. Rensink and J. Warmer, Eds., ECMDA-FA 2006, LNCS 4066, 2006, pp. 376-390.

[69] P. Alves, A. Santos, E. Figueiredo and F. Ferrari, "How Do Programmers Learn AOP? An Exploratory Study of Recurring Mistakes," *Proceedings of 5th Latin-American Workshop on Aspect-Oriented Software Development* (*LA-WASP*.11), Sao Paolo, 26 September 2011, pp. 131-140.

[70] L. Cole and P. Borba, "Deriving Refactorings for AspectJ," *Proceedings of the 4th International Conference on Aspect-Oriented Software Development* (*AOSD*'05), Chicago, 14-18 March 2005, pp. 123-134.

