Mutant Generation for Aspect Oriented Programs

MAYANK SINGH*
Research Scholar
Uttarakhand Technical University, Dehradun, India
mayanksingh2005@gmail.com

SHAILENDRA MISHRA
Professor computer science department
KEC dwarahat, Uttarakhand, India
skmishra1@gmail.com

Abstract: Testing of aspect oriented programs is a new programming paradigm. Many researchers had contributed their research in the field of testing AOP. Mutation testing is an emerging area of research in testing of aspect oriented programming. The effectiveness of mutation testing depends on finding fault types and designing of mutation operators on the basis of faults identified. Therefore the effectiveness of testing depends upon the quality of these mutation operators. We already have the mutation operators for procedural and object oriented languages, but for aspect oriented language only a few researchers have contributed. In this paper we will study in detail about the fault types and related mutation operators for AspectJ language. This paper also proposes the implementation framework to implement these mutation operators automatically.

Keywords: Mutation Testing, Aspect orietend testing, fault based testing

1. INTRODUCTION

Software testing is very crucial part of software engineering. If the testing of software is not appropriate then there is no guarantee of the quality of software product. With the help of testing process we can ensure that software realize all the required functions as well as check the performance of software. The testing process must be done with the intention of finding bugs in the programs. In the software development life cycle, testing activities starts from requirement phase and goes along with all the intermediate process of development. Along with quality process, testing is the only activity which is carried out even after the development. Testing is necessary to develop any software system. Generally testing cost half of the total cost of software development and maintenance. There are two types of software testing first structural testing and second functional testing [2,26,34,46].

Mutation testing or fault based testing is an example of structural testing for assessing the quality of software. In mutation testing we inject the fault in the original program and test same as the original program and compare the result of both programs to check the quality of program. These faulty programs are called mutants. For example, suppose a program is written to add two numbers i.e. c=a+b, for this program mutants are c'=a-b, c'=a*b, c'=a/b. If output of both programs are not same on the same test cases i.e. $c \neq c'$, then the mutant is killed. If all the mutants are killed, functionality of the program is good and test data is adequate. On the other hand, if the outputs of both programs are same that means mutants are alive. Mutants may be alive because of the test data is unable to distinguish the mutants or equivalent mutants.

Every software engineers wants to look for ways to improve the modularity of software. AOP is a new methodology that provides separation of crosscutting concerns (design and requirement elements that affect multiple modules) through the modularization. With the help of AOP, we implement crosscutting concerns in an aspects instead them in the core modules. Aspect weaver, composes the final system by combining core and crosscutting modules through a process called weaving [6,39,47]. Aspect Oriented Programming builds on the top of existing methodologies such as object oriented programming and procedural programming. AOP had some myths like AOP doesn't solve any new problem, AOP promotes sloppy design, it breaks the encapsulation, and all the testing techniques cannot be applied on AOP.

AspectJ is an aspect oriented language which is an extension to the Java programming language. AspectJ is easy to understand for java programmers because it uses Java as base language and provide some benefits to the language. An AspectJ compiler produces class files that conform to the Java byte code specification, any Java virtual machine can execute these files [6,7,8,9]. With the help of AspectJ language we can implement Join Points- predictable execution point, Pointcut- to select the join points, advice-an code execution point, Intertype declaration – to add attributes and methods to previously established classes and aspect – an analogy to encapsulate all the above points into java class [1,3,7].

Testing is big an issue in aspect oriented programming. In this paper we attempt to resolve this issue with the help of mutation testing techniques. Here we classify the possible faults which can occur in AOP. Further we will design corresponding mutation operators to resolve the testing issues with aspect oriented programs. We also propose the framework for implementation of mutation operators.

The rest of the paper is organized as follows: Section 2 describe in detail the related work for fault based testing. Section 3 describes the classification of faults. Mutation operators have been describing in section 4. Section 5 describes the implementation architecture. Conclusion and future work is discussed in section 6.

2. RELATED WORK

Ferrari et al. identified some fault types for aspect oriented programs that are extend in this paper. They have also identified a set of mutation operators related to the faults. They define the operators for aspectJ language and propose generalize mutation operators for other aspect oriented languages [5].

Yves presents a tool named AjMutator for mutation analysis of point-cut descriptor. In this paper they implement pointcut related mutation operators which have been identified in the previous research. This tool leverages the static analysis by the compiler to detect the equivalent mutants automatically [21].

Romain Delamare proposes a test driven approach for the development and validation of pointcut descriptor. They designed a tool named Advice Tracer which is used to specify the expected joinpoints. To validate the process, they also develop a tool that systematically injects faults into pointcut descriptors [50].

Alexander identifies key issues related to the systematic testing of aspect oriented programs. They develop a candidate fault model for aspect oriented programs and derive testing criteria from the candidate fault model [12,13,47].

Prasanth proposes a framework which automatically finds the strength of pointcuts and determines the different versions of the expression to choose the correct strength by the developer. Based on similarity measure, the framework automatically selects and rank different versions of pointcut expression [10,23].

Xie and Zhao developed a framework, named Aspectra, which automatically generate the test inputs for testing aspectual behavior. Aspectra defines and measures aspectual branch coverage [49].

Wedyan and Ghosh present a tool for measuring joinpoint coverage from per advice and per class. This tool is based on AspectJ and Java bytecode. This tool implements the framework which is given by Xie and Zhao [11].

Mortensen and T. Alexander use the static analysis of an aspect with in a system to choose the white box criteria for an aspect such as statement coverage, context coverage and def-use coverage. They also provide a set of mutation operators related to pointcut and aspect precedence [12].

3. CLASSIFICATION OF FAULTS

To distinguish the programs from its mutants, we need effective test cases to find faults in the program. Like other testing techniques, the efficiency of mutation testing depends on finding faults. Any mutation system can have these faults and this paper attempts to identify almost all the faults from such mutation system which is designed to represent these faults. These faults are implemented in the form of mutation operators. Effectiveness of mutation testing is depends on the quality of mutation operators. Mutation testing is not new but with respect to AOP it is new.

AOP have many new features such as pointcut, joinpoint, advice, inter type declaration, aspect and weaving. These new features of AOP introduce the potential of new faults. Previously stated faults about these features are not sufficient, so we have to identify new faults to complete the qualitative mutation testing.

Some of Java related faults can be used in finding the faults of aspect oriented programs because AspectJ program uses Java language for the base program. There are only two researchers named Baekken and Ferrari, who identified fault types and related set of mutation operators [5,24]. All of these fault types focus on the characteristics and structure of AspectJ language. This paper introduces a new set of fault types and mutation operators with the inclusion of all previously stated fault types and mutation operators. Previously stated mutation operators do not handle several fault types and all AOP features.

Faults can be classified on the basis of our exhaustive survey on testing aspect oriented programming [4,5,7,10,11,12,13,14,17,21,24]. Our analysis is based on fault models, bug patterns, pointcut descriptor, and fault classifications [35,38,45,47,49]. In AspectJ language, we can find faults in a program with woven aspect i.e. a fault may exist in the base program that is not affected by the woven aspect or fault can exist in the aspect code [13,51]. Faults can be classified on the basis of pointcut, advice, java program, and intertype declaration and weaving. We have identified some new faults which are given below:

- Visibility of joinpoints or pointcut selection fault because pointcut expression selects joinpoint as it was supposed to or was not selected and neither supposed to or both ignored and unintended joinpoints or selects only ignored joinpoints or selects only unintended joinpoints.
- Faults during combining individual pointcut in conditional operators
- Incorrect use of methods, type in pointcut expression
- Use of wrong filed or constructor pattern in pointcut expression
- Use of wrong primitive pointcut descriptor
- Wrong matching based on exception throwing patterns.
- Use incorrect method name in introduction
- Inconsistent method introduction overridden
- Ripple effect production in the control flow
- Wrong changes in polymorphic calls
- Wrong changes in data dependency
- Wrong changes in control dependency
- Failure to execute the proper context
- Wrong advice precedence
- Incorrect changes in inheritance hierarchy
- Modifying variables and statements
- Inconsistency in interface on dynamic binding
- Incorrect use of advice specification
- Fault in advice code to meet its post-condition
- Wrong introduction of interface
- Addition of members at wrong places in hierarchy
- Adding wrong intertype members
- Static invariants violations occur because of weaving
- Incorrect weaving of around advice
- Failure to establish state invariants
- Incorrect or missing proceed in around advice
- Failure due to unable to find required joinpoints from base code
- Fault due to member variable declaration with parent class

4. MUTATION OPERATORS FOR ASPECTJ

There are mainly four types of mutation operators available namely pointcut related operators, advice related operators, waving related operators and base Java programs related operators. Due to space, the description of these mutation operators are not given here but in the next paper we will provide the implementation details with full

description of all mutation operators. On the basis of fault types, specified in the previous section, the mutation operators are as follows:

- PPCM - Incorrect change of primitive pointcut by call to execution or vice versa of methods and constructors
- PPCC - Incorrect change of primitive pointcut by Initialize to reinitialize or vice versa of constructors
- PPCF - Incorrect change of primitive pointcut by cflow to cflowbelow or vice versa
- PPCT - Incorrect change of primitive pointcut by this to target or vice versa
- PPTA - Incorrect change of primitive pointcut by target to args or vice versa
- PPTW - Incorrect change of primitive pointcut by this to within or vice versa
- PPWC - Incorrect change of primitive pointcut by cflow to within or vice versa
- PPSG - Incorrect change of primitive pointcut by set to get or vice versa
- PNUD - Use incorrect user defined pointcut name
- PPFW - Pointcut positive fault due to the use of wildcard
- PNFW - Pointcut negative fault due to the use of wildcard
- PWBI - Wrong Boolean expression for if pointcut
- PICO - Incorrect use of pointcut composition operators like change OR with AND operators or vice versa
- PCON - Use of NOT composition operator where it should not be used or vice versa
- BPCO - Incorrect use of composition operators individually like more than one args, target and this pointcut composed with AND operators. Change this composition operator with other composition operator
- BMAS - Wrong introduction of methods as abstract with synchronized, final or static
- BCAS - Incorrect introduction of constructor as abstract, static, final, volatile or transit
- BPKT - Wrong use of this keyword in base program
- BPDM - Incorrect deletion of member variable initialization
- BPDP - Incorrect declaration of member variable in parent class
- ACAB - Wrong changes in advice from before to after or vice versa
- AIDP - Incorrect deletion of proceed statement
- PMDP - Missing or deletion of pointcut parameters
- PCDP - Incorrect changes in the parameter list of pointcut descriptor
- ASPD - Deletion of aspect precedence
- BCFW - Unintended control flow execution by changing warning or error statement

5. PROPOSED FRAMEWORK

In this paper we are proposing a framework for implementation of these fault types and mutation operators. The proposed framework attempts to automate the test data generation process. Proposed framework takes AspectJ and Java files as an input. Parse the Aspect file to find the pointcuts and advice code. Then according to the pointcut expression, we have to identify the joinpoints from the base Java code. Compile both files with ajc compiler and add the woven code into a single Java file. After this, we have to decide the testing criteria for mutation testing. Then on the basis of fault types and mutation operators, mutants have been generated and applied according to the need.

Store all the tested data into database. On the basis of first testing data, we have decided the baseline for other iterations and execute the analysis process to check whether mutants are alive or killed. Decision about the status of mutants depends on the output data that are generated from both programs i.e. original program and mutant program. On the basis of final report we find the cost of mutation testing. This framework also finds the quality of the test cases and the mutants. The effectiveness of framework depends on the generation of fault types and mutation operators. The mutation based test case generation process is shown in figure 1.

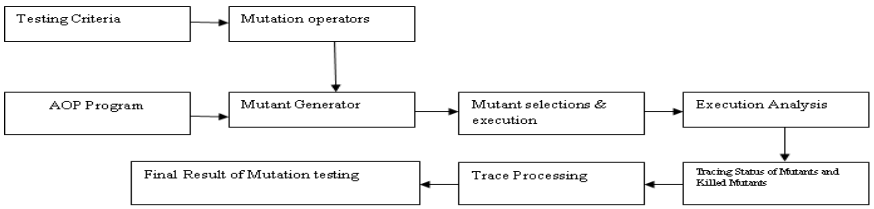

Figure 1: Mutation based test case generation

Test execution is done on the basis of generated test cases. For mutation testing at unit level, we have to identify the test objects through program slicing and then analyze the test outcome with expected outcome. Equivalent mutants are identified and killed with the help of these test objects and test execution process. Process of test execution is shown in figure 2.

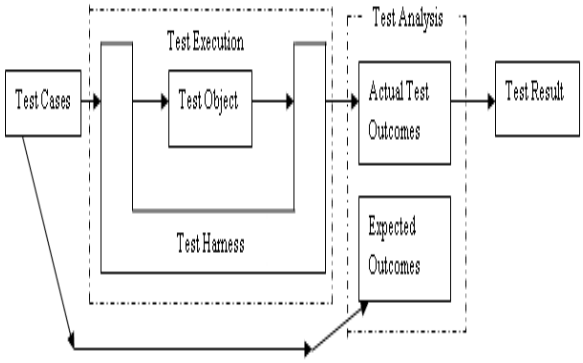

Figure 2: Test Execution Overview

6. CONCLUSION AND FUTURE SCOPE

In this paper we present fault types and mutation operators for mutation testing related to aspect oriented programs. The operators are based on AspectJ language which is most acceptable language for aspect oriented programming. These fault types identified from the characteristics of AspectJ language with Java language. These mutation operators are based on an exhaustive list of aspect oriented faults. This provides a way to improve the efficiency and reliability of aspect oriented software. In this paper we proposed the implementation framework to execute the test data of mutation operators.

Our future scope is to identify some new mutation operators and implement these operators. Our next aim is to develop an automated tool to test these mutation operators as well as generate test cases automatically. We also want to check the quality of the test data to confirm the effectiveness of aspect oriented software.

References

[1] AOSD-Europe Project Home Page, 2007. http://www.aosd-europe.net/ (accessed 05/12/2010).

[2] JBoss AOP Reference Documentation, 2007. http://labs. jboss.com/jbossaop/docs/index.html (accessed 05/12/2010).

[3] The AspectJ Project, 2007. http://www.eclipse.org/aspectj/(accessed 021/11/2010).

[4] R. Agrawal et al. Design of mutant operators for the C programming language. Report SERC-TR41-P, S.E. ResearchCenter, Purdue University, West Lafayette-USA, 1989.

[5] F.C. Ferrari, J.C. Maldonado, and A. Rashid. Mutation testing for aspect oriented programs. In ICST'08, pages 52-61. IEEE Computer Scociety 2008.

[6] AspectJ in Action: Practical Aspect-Oriented Programming, Manning Publications (July 1, 2003).

[7] Mastering AspectJ: Aspect-Oriented Programming in Java, Wiley; 1st edition (March 7, 2003).

[8] Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools, Addison-Wesley Professional (December 24, 2004)

[9] Aspectj Cookbook, O'Reilly Media; 1 edition (December 20, 2004)

[10] Prasanth Anbalagan, Automated Testing of Pointcuts in AspectJ Programs, in OOPSLA'06, October 22–26, 2006, Portland, Oregon, USA.

[11] Fadi Wedyan, Sudipto Ghosh, A Joinpoint Coverage Measurement Tool for Evaluating the Effectiveness of Test Inputs for AspectJ Programs, IEEE Computer Society, 2008.

[12] Michael Mortensen and Roger T. Alexander, Adequate Testing of Aspect-Oriented Programs, Technical report CS 04-110, December 2004, Colorado State University, Fort Collins, Colorado, USA.

[13] R. T. Alexander, J. M. Bieman, and A. A. Andrews. Towards the systematic testing of aspect-oriented programs. Report CS-04-105, Dept. of Computer Science, Colorado State Uni-versity, Fort Collins-USA, 2004.

[14] Hierons, R., Harman, M. & Danicic. S. "Using Program Slicing to Assist in the Detection of Equivalent Mutants". Journal of Software Testing, Verification and Reliability, 9(4), 233-262, 1999..

[15] Offutt, J. & Pan, J. "Automatically Detecting Equivalent Mutants and Infeasible Paths". The Journal of Software Testing, Verification, and Reliability, Vol 7, No. 3, pages 165--192, September 1997.

[16] King, K. N. & Offutt, J. "A Fortran Language System for Mutation-Based Software Testing". Software Practice and Experience, 21(7):686--718, July 1991.

[17] Ma, Y. S., Offutt, J. & Kwon, Y. R. "MuJava: An Automated Class Mutation System". Journal of Software Testing, Verification and Reliability, 15(2):97-133, June 2005..

[18] Ma, Y. S., Kwon, Y. R. & Offutt, J. "Inter-Class Mutation Operators for Java". Proceedings of the 13th International Symposium on Software Reliability Engineering, IEEE Computer Society Press, Annapolis MD, November 2002, pp. 352-363.

[19] Ma, Y. S. & Offutt, J. "Description of Method-level Mutation Operators for Java". November 2005.

[20] Ma, Y. S. & Offutt, J. "Description of Class-level Mutation Operators for Java". November 2005.

[21] AjMutator: A Tool For The Mutation Analysis Of AspectJ Pointcut Descriptors: Romain Delamare, Benoit Baudry and Yves Le Traon, IEEE International Conference on Software Testing Verification and Validation Workshops 2009.

[22] An Analysis and Survey of the Development of Mutation Testing: Yue Jia and Mark Harman, Technical Report- CREST CENTRE, KING'S COLLEGE LONDON. TR-9-06

[23] P. Anbalagan and T. Xie. Efficient mutant generation for mutation testing of pointcuts in aspect-oriented programs. In Mutation'2006, pages 51–56. Kluwer, 2006.

[24] J. S. Bækken. A fault model for pointcuts and advice in AspectJ programs. Master's thesis, School of Electrical Engineering and Computer Science, Washington State University, Pullman-USA, 2006.

[25] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Toward the determination of sufficient mutant operators for C. Soft. Testing, Verif. and Reliability, 11(2):113–136, 2001.

[26] M. Ceccato, P. Tonella, and F. Ricca. Is AOP code easier or harder to test than OOP code? In WTAOP'2005, 2005.

[27] R. Chitchyan, A. Rashid, P. Rayson, and R. Waters. Semantics-based composition for aspect-oriented require- ments engineering. In AOSD'2007, pages 36–48. ACM Press, 2007.

[28] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. PolyAML: A polymorphic aspect-oriented functional programming language. In ICFP'2005, pages 306–319. ACM Press, 2005.

[29] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface Mutation: An approach for integration testing. IEEE Transactions on Soft. Eng., 27(3):228–247, 2001.

[30] M. Gong, V. Muthusamy, and H. Jacobsen. Aspect-Oriented C tutorial, 2006. http://research.msrg.utoronto.ca/ACC/ Tutorial (accessed 23/08/2007).

[31] R. A. DeMillo. Hints on test data selection: Help for the practicing programmer. IEEE Computer, 11(4):34–43, 1978.

[32] M. Eaddy, A. Aho, W. Hu, P. McDonald, and J. Burger.Debugging aspect-enabled programs. In Software Composition'2007, 2007.

[33] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. IEEE Transactions on Soft. Eng., 17(9):900–910, 1991.

[34] W. Harrison, H. Ossher, and P. L. Tarr. General composition of software artifacts. In Software Composition'2005, pages 194–210 (LNCS v.4089). Springer-Verlag, 2006.

[35] O. A. L. Lemos, F. C. Ferrari, P. C. Masiero, and C. V. Lopes. Testing aspect-oriented programming pointcut descriptors. In WTAOP'2006, pages 33–38. ACM Press, 2006.

[36] A. P. Mathur and W. E. Wong. An empirical comparison of data flow and mutation based test adequacy criteria. Soft. Testing, Verif. and Reliability, 4(1):9–31, 1994.

[37] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In AOSD'2003, pages 90–99. ACM Press, 2003.

[38] S. Zhang and J. Zhao. On identifying bug patterns in aspect-oriented programs. In COMPSAC'2007, pages 431–438. IEEE Computer

[39] G. Kiczales et al. Aspect-oriented programming. In ECOOP'1997, pages 220–242 (LNCS v.1241). Springer Verlag, 1997.

[40] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In ICSM'2005,pages 653–656. IEEE Computer Society, 2005.

[41] D. Xu andW. Xu. State-based incremental testing of aspect-oriented programs. In AOSD'2006, pages 180–189. ACM Press, 2006. Society, 2007.

[42] A. J. Offutt, J. Voas, and J. Payne. Mutation operators for Ada. Report ISSE-TR-96-06, Dept. Inf. and Soft. Systems Eng., George Mason Universit, Fairfax-USA, 1996.

[43] 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, 80(6):862–882, 2007.

[44] S. Soares, P. Borba, and E. Laureano. Distribution and persistence as aspects. Software - Practice & Experience,36(7):711–759, 2006.

[45] A. van Deursen, M. Marin, and L. Moonen. A systematic aspect-oriented refactoring and testing strategy, and its application to JHotDraw. Report SEN-R0507, Stichting Centrum voor Wiskundeen Informatica, Netherlands, 2005.

[46] R. Johnson et al. Spring - Java/J2EE application framework. Reference Manual Version 2.0.6, Interface21 Ltd., 2007.

[47] M. Mortensen and R. T. Alexander. An approach for adequate testing of AspectJ programs. In WTAOP'2005, 2005.

[48] W. E.Wong. On Mutation and Data Flow. PhD thesis, Dept.of Computer Science, Purdue University, USA, 1993.

[49] Tao Xie and Jianjun Zhao, A Framework and Tool Supports for Generating Test Inputs of AspectJ Programs, AOSD 06, March 20–24, 2006, Bonn, Germany.

[50] Romain Delamare, Benoit Baudry and Sudipto Ghosh, A Test-Driven Approach to Developing Pointcut Descriptors in AspectJ, Proceeding, ICST '09 Proceedings of the 2009 International Conference on Software Testing Verification and Validation.

[51] Freddy Munoz, Benoit Baudry, Romain Delamare, Yves Le Traon, Inquiring the Usage of Aspect-Oriented Programming: An Empirical Study, European project DiVA (EU FP7 STREP), Software Maintenance, 2009. ICSM 2009.