Testing Aspect Oriented Software Using UML Activity Diagrams

Charnpreet Kaur, Sushil Garg
CSE Deptt.RIMT-IET, Mandi Gobindgarh, HOD CSE Deptt.RIMT-IET, Mandi Gobindgarh

Abstract

Aspect oriented programming (AOP) is an extension to object oriented programming (OOP). Aspect oriented programming supports the separation of crosscutting concerns. AOP is a software engineering paradigm that gives new types of constructs such as advice, join points, point cut and aspect in order to improve the separation of concerns. AOP new constructs brings new types of faults incorrect advice, point cut and aspect precedence. In this paper we test the Aspect specific faults with UML activity Diagrams and check that it conforms to its expected crosscutting behavior. This approach focus on integration of crosscutting concerns to primary concern and generates test sequence based on interaction between aspect and primary models and verifies the execution of selected sequence.

1. Introduction

Aspect oriented programming is a technology that supports separation of crosscutting concerns [1], [2] (i.e. functionality that tends to be tangled with and scattered through the rest of the system). The crosscutting mechanism of aspects frees the programmer from interweaving different concern (goals, concepts or area of interest). The concept of AOP supports modularization of different concerns while in traditional approach there are concerns that are clearly mapped to isolated modules of implementation for instance concern such as Access control, synchronization policies and logging tend to be tangled with and scattered throughout the basic modules of implementation (dominated base code) these concerns are called crosscutting concerns.

AOP Supports the implementation of separate modules called aspects that have ability to cut across other modules, adding behavior that would otherwise spread throughout the base code. AOP languages defined new features due to this new types of faults occurs (1) incorrect point cut (2) Incorrect Advice (3) incorrect aspect precedence.

A join point is a well defined point in the execution of the program such as method calls a constructor invocation or a variable access. A point cut is a set of pattern that is used to select join points. Advice is a method like construct that contains additional behavior to be added at the matched join points. Advices represent a fragment of control and data that must be added to the body of existing method. Aspect is a construct that encapsulates a crosscutting concern. Aspect weaving is the process by which behavior on aspects are merged to the original code. In aspect oriented programming the behavior of aspect are merged to the original code. The process of weaving cause a lots of problems in the testing process because it is very difficult to predict exactly how this code will weave in the other code what kind of changes will it introduce, what kind of dependencies or change in the already existing dependencies will be introduced. Will the base code be changed by weaving the code and how all the changes will influence the behavior of the system?

In this paper we present a UML activity diagram based approach to test aspect oriented program or verify that it conforms to its expected behavior. This approach focuses on integrating one or more aspects to primary model or generates test sequences. In this approach firstly make activity diagram of basic model then generate the test sequences for basic model secondly weave the aspects to basic model and generate integrated model then generate the test sequences for integrated model and finally verify that actual behavior match to the expected behavior.

2. Related work

AOP provides a flexible mechanism for modularizing crosscutting concerns [3] it raises new challenges for testing aspect oriented programs. Alexander et al. [4] have proposed a fault model for aspect-oriented programming, which includes six types of faults: incorrect strength in point cut patterns, incorrect aspect precedence, and failure to three establish post conditions, failure to preserve state invariants,

incorrect focus of control flow, and incorrect changes in control dependencies (Alexander et al., 2004). This fault model has not yet constituted a fully developed testing approach. While some faults (e.g., incorrect point cut strength and incorrect aspect precedence) are undoubtedly useful for developing testing tools and determining coverage strategies, others are subtle. For example, failures to establish post conditions or preserve state invariants assume that the contract of classes should be enforced by aspects at the design level.

Zhao in [5] has proposed a data flow based unit testing approach for aspect oriented programs. This approach tests two types of units for an aspect-oriented program, i.e., aspects that are modular units of crosscutting implementation of the program, and those classes whose behavior may be affected by one or more aspects. For each aspect or class, this approach performs three levels of testing, i.e., intra-module, inter module, and intra-aspect or intra-class testing. This approach can handle unit testing problems that are unique to aspect-oriented programs. This approach uses control flow graph to compute def-use pairs of an aspect or class being tested and use such information to guide the selection of tests for the aspect or class. Zhao and Rinard [6] have also exploited system dependence graphs to capture the additional structures in aspect-oriented features such as join points, advice, aspects, and interactions between aspects and classes. In this approach, control flow graphs are constructed at both system and module level, and test suites are derived from control flow graphs. There is no any fault model is targeted to help detect most likely faults.

Xu et al. proposed different approaches for testing aspect-oriented programs [7], [8], [9]. They proposed in [7] a state-based approach for unit testing aspect-oriented programs. Their approach is based on a model called Aspectual State Model (ASM) that is an extension to the testable FREE (Flattened Regular Expression) state model to capture the impact of aspects on the state models of classes. Once the ASM is created, it can be transformed into a transition tree, which implies a test suite for adequately testing object behavior and interaction between classes and aspects in terms of message sequences. In [8] they presented an incremental testing approach for aspect-oriented programs. The main idea of this approach is to reuse the base class tests for testing aspects according to the state-based impact of aspects on their base classes. In particular, an extended state model for capturing the impact of aspects on the state transitions of base class objects as well as an explicit weaving mechanism for composing aspects into their base models is presented. In addition, several rules have been proposed for maximizing reuse of concrete base class tests for aspects. They also proposed in [9] a state-based approach for testing integration aspects. They indicate that an aspect integrating separated concerns, like other aspects, can contain various programming faults. Thus, they exploit an aspect-oriented state model to specify integration aspects. By composing the state models of aspects and classes, they are able to generate test cases for integration aspects from their state models. In addition, Xu et al. proposed in [11], [12] an approach based on different UML design models (class diagrams, aspect diagrams and sequence diagrams) to derive test cases covering the interactions between aspects and classes. Liu and Chang in [13] proposed a state-based testing approach for AOP programs. The approach considers the state-based behavior changes introduced by different advices in multiple aspects. A test model is suggested to depict the state based behavior of aspect-oriented program after aspect weaving. Based on this model, test cases can be derived in order to uncover the potential state behavior errors in the AOP programs. Badri et al. [14] presented a state- based unit testing technique for aspect-oriented programs and associated tool that focuses on the integration of one or several aspects to a class. It supports both the generation and verification of test sequences and its objective is to ensure that the integration is done correctly, without altering the original behavior of the classes. The above works focus on the behavior of a class where one or more aspects are weaved. Our research is related to the integration of one or more aspects to the behavior of a group of objects. We propose an UML activity diagram based approach to testing aspect-oriented programs that is capable reveal some of aspect-specific faults in the early stage of program development. Our work is based on a paper presented by Cui et al. [15] on modeling and integrating aspects with UML activity diagram. We improve this work from the perspective of model-based test sequences generation, and test sequences execution and verification.

3. Overview of this approach

This approach is consists of three steps. The first step is related to building activity diagram of basic model and generating the test sequence of basic model. This step reduces the complexity to eliminate faults that are only related to basic model not related to Aspects. The second phase is to integrating the aspects with the Basic model and generates the test sequences. The third phase consists of verifying the execution of the sequence and compares the actual sequence with the expected sequence.

4. Testing process

Aspect oriented testing with activity diagram integrate the aspect with basic model then generate the test sequences and finally execute the test sequences to verify process.

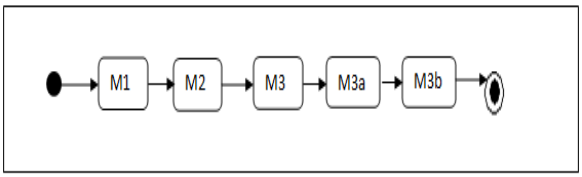
4.1. Aspect oriented Activity diagram

Aspect oriented activity diagram describes the dynamic aspects of the system. Activity diagram is basically a flowchart to represent the flow from one activity to another. Aspect oriented activity diagram motivated to capture important features like join points, point cut and advice etc. Aspect oriented model consist of basic model and aspect model. Basic model is the actual model gives sequence of activity diagram. Aspect model which consist of point cut model and corresponding advice model. A point cut model is used to select join points like nodes edges etc and join point picked from basic model, An Advice model specify additional behavior that added to basic model (before, After etc).

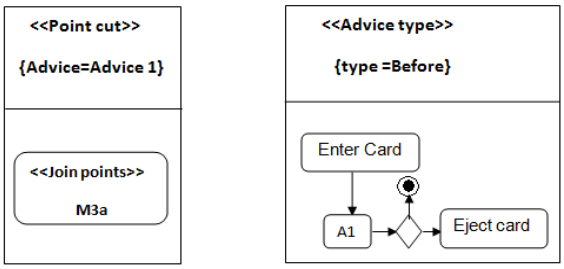
Crosscutting concerns are either sequential or parallel aspects that are running sequentially or in parallel with basic concerns. Sequential aspect in process depends on the result of another process. Parallel Aspects in process running results not influence the other process.

In this Testing Process Figure1 shows simple aspect oriented activity model basic model, Sequential aspect A1and Parallel aspect A2. Aspect A1 in figure 1(b) consist of Point cut model and Advice model. Point cut model select the elements in the Basic model to which the sequential advice A1 will be applied. The point cut model is denoted with <<Point cut>>.The Point cut model gives the Join points. In figure 1(b) model A1 concern is sequential which means advice action A1 is performed before the join point nodes. The advice model is denoted with <<Advice>>.The tagged value type which indicates the type of advice is "Before". In Advice model Enter Card or Eject Card show the flow of basic model.

The aspect A2 in Figure 1(c) consist of point cut model and advice model constructed to select the elements in the basic model to which parallel advice is applied. Advice A2 concern as parallel Advice and action A2 runs parallel with basic model. Aspect oriented activity model essentially depend on the weaving mechanism that composes aspect models with the basic model. The result of this composition is integrated model. Integrated model is prepared by finding join points in primary model, initializing advice model, and weaving advices into basic model. Figure2 is the integrated model after weaving the advice with basic model. In this model Advice1 is inserted before the "M3a" node and Advice2 is inserted after the "M3b".

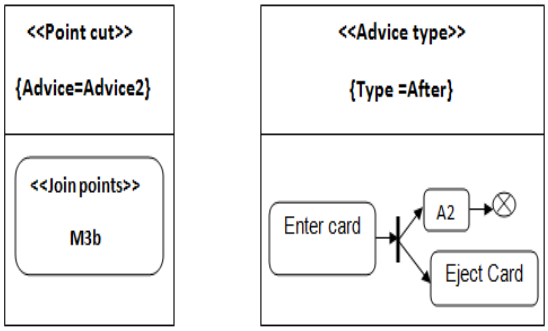


(a) Basic model of activity diagram



Point cut1 **Advice1**

(b) Aspect1



Point cut2 **Advice2**

(c) Aspect2

Figure1. Aspect oriented activity diagram

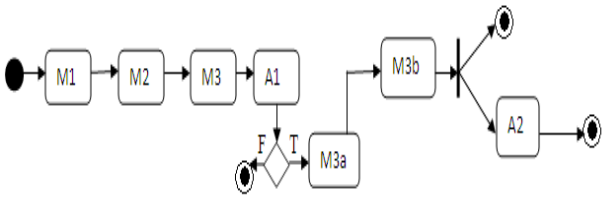


Figure2. Integrated model

4.2. Test sequence generation

We start by generating the test sequence of basic concern and test this separately. This process reduces the complexity of testing process and removes the faults related to Basic concern. Table1 shows the test sequence of Basic model depicted in Figure1 (a)

Table1. Basic test sequence

Number	Test Sequence
1	M1→M2→M3→M3a→M3b

Table2. Integrated test sequence

Number	Test Sequence
1	M1→M2→M3→A1
2	M 1→M2→M3→A1→M3a→M3b
3	M 1→M2→M3→A1→M3a→M3b→A2

5. Case study

The above approach is applied to the ATM system and test the incorrect point cut, incorrect advice and incorrect aspect precedence. We take simple example of ATM system in which user validation and withdraw money features are added using aspects where withdraw money depend on the user validation. ATM system consists of:

- Customer has ATM card to access the account.
- ATM machine make all transactions and show balance amount to user.
- Bank check for validate user with the help of pin no. Bank also checks for sufficient amount to withdraw
- The aspects of the system are user validation to check for validity of user. After checking for valid user pin number user withdraws money. User validation follows withdraw money.

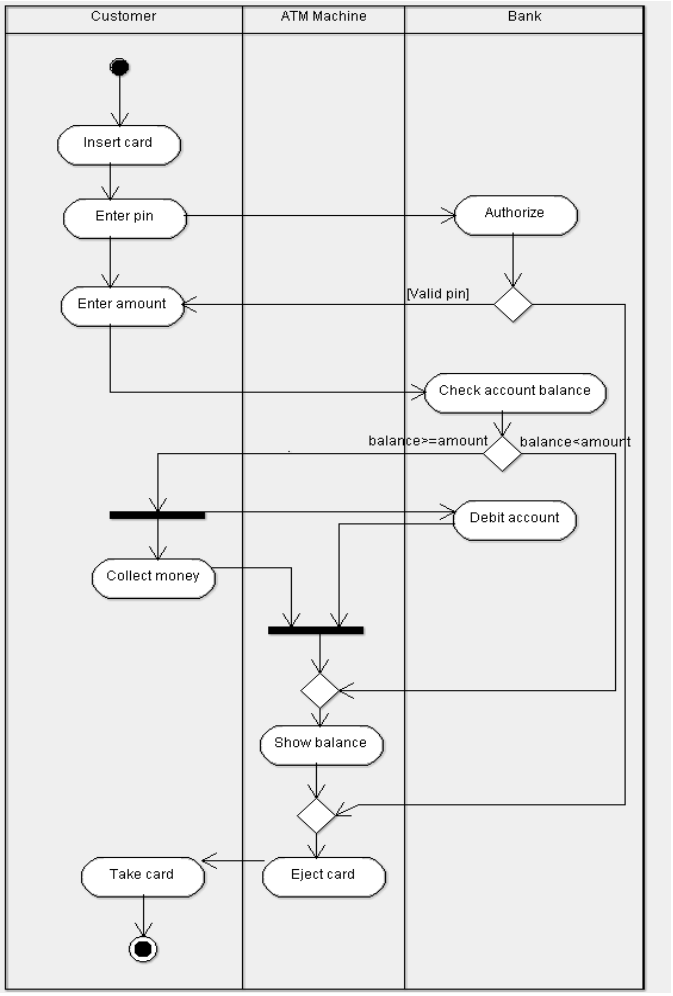


Figure3. Diagram of ATM system

In table 3 various faults like incorrect advice type, weak or strong point cut strength and incorrect aspect precedence. Our experiment results reveal these types of faults. In the table 3 shows example of each of target fault type. Each row is for special type faults it includes:

- The specification of advice type, point cut and aspect precedence.
- The Expected method sequences.
- The actual implementation of advice type, point cut and aspect precedence.
- The actual method sequences.

Table3. Shows the revealing of various faults

Type of fault	Model		Implementation	
	Advice type /Point cut pattern/ Aspect precedence	Expected sequence	Advice type /Point cut pattern/ Aspect precedence	Actual sequence
Incorrect advice type	After/ ATM(float withdraw money)/ NA	Customer. Insert card()-> Customer. Enter pin()-> Bank. Authorize()-> Customer. Enter amount()-> Bank. Check account bal()-> Customer. Collect money()-> Bank. Debit account()-> ATM. Show balance()-> ATM. Eject card()-> Customer. Take card()	Before/ ATM(float withdraw money)	Customer. Insert card()-> Customer. Enter pin()-> Bank. Authorize()-> Bank. Check account bal()-> Customer. Enter amount()-> Customer. Collect money()-> Bank. Debit account()-> ATM. Show balance()-> ATM. Eject card()-> Customer. Take card()
Weak point cut	After/ ATM(float withdraw money)/ NA	Customer. Insert card()-> Customer. Enter pin()-> Bank. Authorize()-> Customer. Enter amount()-> Bank. Check account bal()-> Customer. Collect money()-> Bank. Debit account()-> ATM. Show balance()-> ATM. Eject card()-> Customer. Take card()	After/ ATM(float withdraw money*)/ NA	Customer. Insert card()-> Customer. Enter pin()-> Bank. Authorize()-> Customer. Enter amount()-> Customer. Collect money()-> Bank. Debit account()-> ATM. Show balance()-> ATM. Eject card()-> Customer. Take card()
Strong point cut	After/ ATM(float withdraw money)/ NA	Customer. Insert card()-> Customer. Enter pin()-> Bank. Authorize()-> Customer. Enter amount()-> Bank. Check account bal()-> Customer. Collect money()-> Bank. Debit account()-> ATM. Show balance()-> ATM. Eject card()-> Customer. Take card()	After/ ATM(float withdraw money not updated)/ NA	Customer. Insert card()-> Customer. Enter pin()-> Bank. Authorize()-> Bank. Check account bal()-> Customer. Enter amount()-> Customer. Collect money()-> ATM. Show balance()-> ATM. Eject card()-> Customer. Take card()
Incorrect precedence	NA/ NA/ Debit account Show balance	Customer. Insert card()-> Customer. Enter pin()-> Bank. Authorize()-> Customer. Enter amount()-> Bank. Check account bal()-> Customer. Collect money()-> Bank. Debit account()-> ATM. Show balance()-> ATM. Eject card()-> Customer. Take card()	NA/ NA/ Show balance Debit account	Customer. Insert card()-> Customer. Enter pin()-> Bank. Authorize()-> Customer. Enter amount()-> Bank. Check account bal()-> Customer. Collect money()-> Bank. Debit account()-> ATM. Show balance()-> ATM. Eject card()-> Customer. Take card()

In the first row of table the expected sequence is different from actual sequence. The difference in sequences helps us to reveal faults. In this row advice is changed from after (in expected) to before (in

actual). The second row describes a weak point cut fault. Weak point cut means point cut that is not too necessarily needed. In this row check account balance is not necessary to collect money. Third row describes strong point cut that is necessarily needed. In this row debit account is necessary to show the balance.

6. Conclusion and future work

In this paper we describe the testing aspect oriented program with UML activity diagram. This approach helps the tester to reveal the various types of faults such as incorrect advice type, strong or weak point cut, and incorrect aspect precedence. In this strategy tree main steps are (1) Make activity diagram of basic concern and generating the test sequences. This step reveals the various faults present in basic model not in aspect (2) Make aspect model and integrate them in to basic model with this increment way we reduce the complexity of test. (3) This step verifies the generated sequences that implementation conforms to its specification. This approach manually generate the test sequence for future work this manual approach can be automated or enhanced with automated test sequence generation.

7. References

[1] R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, "Aspect-Oriented Software Development", Addison-Wesley Professional Boston, 2004.

[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier, and J. Irwin, 1997 "Aspect oriented programming" In Proc. of ECOOP'97, LNCS 1241, pp. 220-242.

[3] The Aspectj Team. The Aspectj Programming Guide. August 2001

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

[5] J. Zhao, "Data-flow-based unit testing of aspect-oriented programs", In Proc of the 27th Annual IEEE International Computer Software and Applications Conference (COMPSAC'03), pp.188-197, 2003.

[6] J. Zhao and M. Rinard, "System dependence graphs construction for aspect-oriented programs", MIT-LCS-TR -891, Laboratory for Computer Science, MIT, March 2003.

[7] 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, pp. 366-371, 2005.

[8] 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, pp. 180-189, 2006.

[9] W. Xu, and D. Xu, "State-based testing of integration aspects", In Proceedings of the 2nd Workshop on Testing Aspect-Oriented Programs, pp. 7-14, 2006.

[10] R. V. Binder, "Testing Object-Oriented Systems: Models, Patterns, and Tools", Addison-Wesley Professional, Boston, 2000.

[11] W. Xu, D. 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, May 2008.

[12] W. Xu, and D. Xu, "A model-based approach to test generation for aspect-oriented programs", AOSD 2005 Workshop on Testing Aspect-Oriented Programs, Chicago, 2005.

[13] C. H. Liu, and C. W. Chang, "A State-Based Testing Approach for Aspect-oriented Programming", In Journal of Information Science and Engineering , pp. 11-31, 2008.

[14] B. Badri, L. Badri, M. B. Fortin, "Automated State-Based Unit Testing for Aspect-Oriented Programs: A Supporting Framework", In Journal of Object Technology, vol. 8, no. 3, pp. 121-126, 2009.

[15] Z. Cui, L. Wang and X. Li, "Modeling and integrating aspects with UML activity diagrams", Proceedings of the 2009 ACM symposium on Applied Computing, 2009.

[16] M. Mortensen, and R. Alexander, "Adequate Testing of Aspect-Oriented Programs", Technical report CS 04-110, Colorado State University, Fort Collins, Colorado, USA, December 2004.

[17] Offut, J., Xiong, Y., and Liu, S., "Criteria for Generating Specification-based Tests", In Engineering of Complex Computer Systems, ICECCS '99, 1999.

[18] C. Chavez, and C. Lucena, 2002. A meta-model for aspect-oriented modeling, The Workshop on Aspect-Oriented Modeling with UML.

[19] A. Andrews, R. France, S. Ghosh, and G. Craig, "Test adequacy criteria for UML design models" Journal of Software Testing, Verification and Reliability, 13(2):95-127, 2003.

[20] E. Barra, G. Génova, and J. Llorens, (2004)."An approach to aspect modeling with UML 2.0". The Fifth International Workshop on Aspect-Oriented Modeling (AOM'04).

[21] M., and Andrews, A.A., 2004. "Towards the systematic testing of aspect-oriented programs", Technical Report, Colorado State university http:// www.cs.colosate.ed/~rta/publications/CS-04-105.pdf.

[22] AspectJ Web Site, http://eclipse.org/aspectj/.