Celadon: A Change Impact Analysis Tool for Aspect-Oriented Programs

Sai Zhang, Zhongxian Gu, Yu Lin, Jianjun Zhao
School of Software
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai 200240, China
{saizhang,ausgoo,linyu1986,zhao-jj}@sjtu.edu.cn

ABSTRACT

To reduce the manual effort of assessing potential affected program parts during software evolution, we develop a tool, called Celadon, which automates the change impact analysis for AspectJ programs. Celadon is implemented in the context of the *Eclipse* environment and designed as a plugin. It analyzes the source code of two AspectJ software versions, and decomposes their differences into a set of atomic changes together with their dependence relationships. The analysis result is reported in terms of impacted program parts and affected tests. For each affected test, Celadon also identifies a subset of *affecting changes* that are responsible for the test's behavior change. In particular, as one of its applications, Celadon helps facilitate fault localization by isolating failure-inducing changes for one specific affected test from other irrelevant changes .

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.6 [**Software Engineering**]: Programming Environments; D.3.3 [**Programming Language**]: Language Constructs and Feature

General Terms

Algorithms, Languages, Measurement, Reliability

Keywords

Aspect-oriented programming, change impact analysis

1. INTRODUCTION

Change impact analysis is a promising technique for software evolution. It determines the effects of a source editing session and provides valuable feedbacks to the programmers for making correct decisions. It can be used to predict the potential impact of the proposed changes before they are applied, or to estimate the potential side-effect of changes. The information provided by change impact analysis would be very important for developers to eliminate potential risks and improve their productivity in software development.

Aspect-Oriented Programming (AOP) [3] has been proposed as a technique for improving separation of concerns in software design and implementation. An AspectJ program can be divided into two parts: *base code* which includes classes, interfaces, and other language constructs as in Java, and *aspect code* which includes aspects for modelling crosscutting concerns in the programs. With the inclusion of join points, an aspect woven into the base code is solely responsible for a particular crosscutting concern, which raises the system's modularity. However, when implementing changes in AspectJ programs, it involves more complex impacts than in the traditional programming languages:

- The woven aspect may change control and data dependency to the base code. When either changes in base code (like *renaming classes, fields or methods*) or aspect code (like *adding a new pointcut or advice*) occur, the semantics of program can be silently altered.
- The nontrivial combination of source changes may affect other parts of program unexpectedly and indicate potential defects, due to the inherent semantic intricacies of aspectual features (such as *multiple advices per join point*, *dynamic pointcut* et al.).
- Failures of regression tests could either result from changes in base code or a particular aspect, or even the interactions between the base and aspect code. In such cases, examining each place of source modification and pinpointing the few that introduces the failure become a tedious task.

To provide developers with tool support, we extend the concept of atomic change [5] to handle the unique features of AspectJ programs and develop an analysis tool, called Celadon, which helps programmers understand the impact of program changes. Given two versions of an AspectJ program, Celadon decomposes their differences into a set of atomic changes [5, 7] together with their dependence relationships, identifies a subset of regression tests that are impacted by those changes and determines the responsible affecting changes for each affected test. For the updated program version, it also determines the impacted code fragments (at a method level) caused by the applied changes[1].

Celadon allows programmers to observe the affected code fragments after certain changes are applied, to select only the affected tests to rerun during regression testing and then reduce the efforts spent on debugging through isolating failure-inducing changes for one specific affected test from other irrelevant changes.

2. CELADON TOOL

In our previous work [7, 8], we identified a catalog of atomic changes shown in Table 1. Those atomic changes represent the source modifications at a coarse-grained model.

[1]Celadon constructs static AspectJ call graphs for the program and each test as the basis of analysis.

Copyright is held by the author/owner(s).
ICSE'08, May 10–18, 2008, Leipzig, Germany.
ACM 978-1-60558-079-1/08/05.

Abbreviation	Atomic Change Name
AA	Add an Empty Aspect
DA	Delete an Empty Aspect
INF	Introduce a New Field
DIF	Delete an Introduced Field
CIFI	Change an Introduced Field Initializer
INM	Introduce a New Method
DIM	Delete an Introduced Method
CIMB	Change an Introduced Method Body
AEA	Add an Empty Advice
DEA	Delete an Empty Advice
CAB	Change an Advice Body
ANP	Add a New Pointcut
CPB	Change a Pointcut Body
DPC	Delete a Pointcut
AHD	Add a Hierarchy Declaration
DHD	Delete a Hierarchy Declaration
AAP	Add an Aspect Precedence
DAP	Delete an Aspect Precedence
ASED	Add a Soften Exception Declaration
DSED	Delete a Soften Exception Declaration
AIC	Advice Invocation Change

Table 1: A catalog of atomic changes in AspectJ

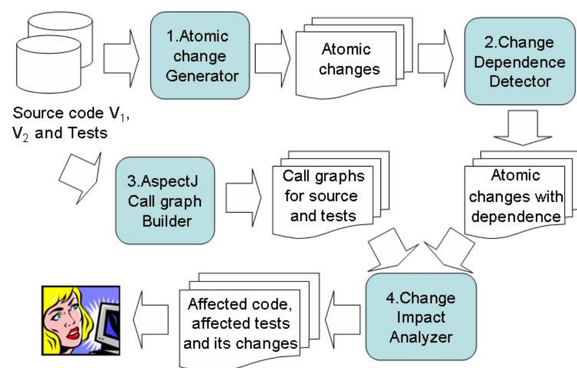

Figure 1: The overview structure of Celadon.

The overview structure of Celadon is shown in Figure 1. Celadon consists of four main components: the atomic change generator, atomic change dependence detector, AspectJ call graph builder, and change impact analyzer.

2.1 Atomic Change Generator

We implement the atomic change generator based on the AspectJ Development Tools (AJDT) [1], which provides plenty of APIs for accessing and manipulating the abstract syntax tree (AST) of an AspectJ program. When comparing two program versions, we use a dynamic programming algorithm [6] to calculate the differences between two ASTs and then generate the corresponding atomic change set. Like the classic LCS problem solution, we first compute the common node pairs between ASTs, then remove those common parts from both versions temporarily. Therefore, the remaining nodes are either new added or deleted during source changes. For the removed common node pairs, we continue to compute the common node pairs in their sub-AST nodes and do the same algorithm recursively. For the **AIC** change, we get the join point matching information when the *ajc* weaver composes the aspects and base code and then calculate the changes.

2.2 Change Dependence Detector

The atomic changes output by the atomic change generator are fed as inputs to the change dependence detector. The change dependence detector uses the semantic dependence rules [8] between atomic changes to determine the dependence relationships. The atomic changes together with their dependence relationships are then passed to the change impact analyzer.

2.3 AspectJ Call Graph Builder

The AspectJ call graph builder constructs call graphs [4] for both edited AspectJ program and its tests. In our implementation, we use the RTA algorithm [2] to construct call graph of the base code. For the aspect code, we consider the advice as a method-like node with matching relationship represented by an edge from the join point. The complete call graph of an AspectJ program is formed after the call graph of aspect code is *connected* into the base code graph using the join point matching information. We treat the anonymous pointcut as a part of advice declaration, thus any changes to the anonymous pointcut will result in the definition change of associated advice. However in AspectJ programs, a dynamic pointcut like *cflow*, *if* and *target* that statically matches a shadow could potentially not match that shadow at run time, it is quite difficult to accurately compute the impact of changing dynamic pointcuts. In Celadon, when constructing the call graph for an AspectJ program, we conservatively assume that for all dynamic pointcuts, whether they match a shadow or not has to be determined at run time. Under this assumption, advices associated with dynamic pointcuts are connected to the corresponding call nodes, which means that changes in this dynamic advice will affect its calling method. Through this way, though approximately, we can safely calculate the affected methods concerning changes in a dynamic advice.

2.4 Change Impact Analyzer

The change impact analyzer in Celadon takes the outputs of change dependence detector and AspectJ call graph builder as inputs. By traversing the call graphs from the changed node, we can easily get all the affected tests with their affecting changes, and affected code fragments in the edited program version. The output of change impact analyzer is displayed in the graphic user interface of Celadon.

Acknowledgements

This work was supported in part by National High Technology Development Program of China (Grant No. 2006AA01Z158), National Natural Science Foundation of China (NSFC) (Grant No. 60673120), and Shanghai Pujiang Program (Grant No. 07pj14058).

3. REFERENCES

[1] AspectJ Development Tools (AJDT). `http://www.eclipse.org/ajdt/`.

[2] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *In OOPSLA '96 Conference Proceedings, San Jose, CA, October 1996*, pages 324–341, 2004.

[3] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. 11th European Conference on Object-Oriented Programming*, pages 220–242. 1997.

[4] N. Li. The call graph construction for aspect-oriented programs. Master's thesis, School of Software, Shanghai Jiao Tong University, March 2007 (in Chinese).

[5] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 432–448, Vancouver, BC, Canada, October 26–28, 2004.

[6] W. Yang. Identifying syntactic differences between two programs. *Software - Practice and Experience*, 21(7):739–755, 1991.

[7] S. Zhang and J. Zhao. Change impact analysis for AspectJ programs. Technical Report SJTU-CSE-TR-07-01, Center for Software Engineering, SJTU, Jan 2007.

[8] S. Zhang and J. Zhao. Locating faults in AspectJ programs. Technical Report SJTU-CSE-TR-07-05, Center for Software Engineering, SJTU, Sep 2007.