Automatic translation from name-based pointcuts to analysis-based pointcuts for robust aspects

Wang, Lin (0910216)

School of Information Science,
Japan Advanced Institute of Science and Technology

August 10, 2011

Keywords: Aspect-Oriented Programming, analysis-based pointcuts, name-based pointcuts, fragile pointcut problem.

Modern software projects are of large scale, often involving tens of thousands of days of work efforts, and millions of lines of code. Moreover our limited minds cannot possibly consider everything and solve everything at once. Therefore, the software is too complex to design merely in a single view, and it is necessary to separate the different concerns in a large piece of software to decompose it into smaller, more manageable units. Separation of concerns(SoC) is an important principle in software engineering, and it has the ability to separate program into distinct features. Without it large software system simply could not be realized. Effective separation of concerns makes a program easier to understand, change and debug. Concerns are used to organize and decompose software into manageable and comprehensible parts. Modularity of programming is a traditional way of achieving separation of concerns. Object-oriented programming (OOP) is a way of modularizing common concerns, classes in object-oriented model perform a single specific function. In many case, we find that many parts of our system have code fragments for logging, persistence, debugging, authorization, tracing, exception handling, and other such tasks. As a result, they have to be coordinated with other functional unites and code scattered usually throughout several functional units exist in their concerns. Thus,

OOP does not do as a good job in these situations. Aspect-Oriented programming (AOP) fills this void. Aspect-Oriented programming provides a solution to the crosscutting problem by supporting the modularization of crosscutting concerns in aspects. In contrast with OOP, AOP is a way of modularizing crosscutting concerns. Crosscutting concern is concern that affect several classes or modules, and the code fragments which implement crosscutting concerns are spread across many units. The modules for the crosscutting concerns are called aspects, aspects encapsulate behaviors that affect multiple classes into reusable modules. AspectJ is the most prevalent and studied in mainstream aspect languages. It is an aspect-oriented programming extension of Java. AspectJ introduces a new concept, join points, and a few new programming constructs, such as pointcuts and advice. A join point is a point in the execution of a program whose behavior can be affected by advice. A pointcut is a construct designed to identify and select join points. Advice defines code to be executed when a join point is reached. Pointcuts match joinpoints, which are well defined points in the execution of a program, so it is a key element in aspect-oriented languages.

Using names and wildcards to capture join points over programs is the most basic way to define the pointcuts. As the program evolves, changes in the structure or naming conventions can make advice accidentally execute an unwanted advices or miss execute some required ones. In other words, there is tight coupling between traditional pointcuts and structure of base code. In this case, when the program evolved, as even a small modification to the program, the pointcut itself have to evolve along with the base-code. This problem is called fragile pointcut problem. Hence it is important to define pointcut in a robust way against program evolution. Robustness means the pointcuts are able to continue to capture the correct set of join points without alters its form in future version of the base-code.

There are many approaches such as Josh, Alpha, Model-based pointcut, Pointcut Rejuvenation etc. that attempt to overcome pointcuts' fragility. These approaches either propose several new pointcut language constructs to improve pointcuts' expressiveness or provide developer some additional information by program analysis. There is, however, no automatic way to translate name-based pointcuts into robust ones, and hence programmers

have to rewrite the existing pointcuts manually. Unfortunately, this is often difficult because new constructs are very different from the original languages. For example, Alpha uses a logic programming language for the specification of pointcuts, it is very different from the name-based pointcuts.

In this dissertation we propose a framework called Nataly, which translates name-based pointcuts into analysis-based pointcuts automatically. Our approach can not only alleviate fragile pointcut problem but also bridge a gap between original name-based pointcut and other robust one. Name-based pointcuts directly use class and method names; they merely check that a called/executed method has the specified name/type. Analysis-based pointcuts are proposed as an approach to overcome the fragility. They use static program analysis rather than names, and match the join points that satisfy the match strategy checked by the analysis. There are three significant components in our framework, namely, relation analyzer, pattern generator and code generator. Given base-code aspects, it first analyzes the program to extract relation maps that represent relationships among fields, methods, classes and interfaces. These relation maps are separated in two types, namely adjacent relation maps and opposite relation maps, respectively. Pattern generator creates relationship trees by using relation maps and join point shadows which correspond to a given pointcut. The roots of the relationship trees are the java elements which associated with join point shadows. Relationship trees represent the behavior of the associated join point shadows which matched by a given pointcut. Intention patterns are extracted from relationship trees and persisted to Tregex format. The purpose of intention pattern is to approximate the essence of the developer's intentions behind the original pointcut. Finally code generators generate code of analysis-based pointcuts by using StringTemplate in our implementation. We implement the Nataly framework for AspectJ in Java.

We also illustrate a case study that evaluates robustness of name-based pointcuts and analysis-based pointcuts in face of seven possible change scenarios to the classic Figure Editor System. It shows that the generated analysis-based pointcuts are more robust than their name-based counterparts against seven program changes, however they still break if

the program changes get complicated. Frankly, we have to declare that our approach can alleviate the problem of fragility, but it cannot solve all the possible instances of fragile pointcut problems currently.

Our approach has several contributions:

- It bridges the gap between traditional name-based pointcut and other robust pointcut language. Our approach is the first one that attempt to translate name-based pointcuts into robust analysis-based pointcuts. The new pointcut languages are very different from the original language, so they are difficult to be written by a programmer who is not familiar with the new pointcut language. Therefore we implement a framework to generate analysis-based pointcut automatically.

- It alleviates fragile pointcut problem by using analysis-bases pointcut. Evaluations of our approach in seven different change scenarios represents that analysis-based points generated by our framework are more robust than its counterpart name-based pointcuts. To five of seven considered scenarios name-based pointcuts are not robust. On the other hand, the solutions based on analysis-based pointcuts are not robust to only one change scenario.

- Our implementation of framework is excellent integration with AspectJ. Analysis-based pointcut in our approach merely relies on existing AOP constructs such as conditional pointcuts. Moreover, the advanced compiler SCoPE can support our analysis-based pointcut.