Improving Transformation of Object-Oriented Program to Aspect-Oriented Program with Static Analysis

by

Lin Wang

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Supervisor: Associate Professor Masato Suzuki

School of Information Science
Japan Advanced Institute of Science and Technology

March, 2015

Abstract

Separation of concerns is expected to be supported by development methodology or programming languages through enabling encapsulation of each different concern in its own unit of modularity. Unfortunately, current object-oriented languages and development methodology fail to provide a complete and effective support for the separation of concerns. Undesirable phenomena such as code scattering and code tangling occur.

Aspect-Oriented Programming (AOP) technique supports the separation of crosscutting concerns. Aspect-orientation proposes a technique to obtain better software modularity in a practical situation where object-oriented development/programming and associated design patterns are not appropriate. There is a prospect of aspect-oriented programming becoming a mainstream technology in the near future. The questions of how to deal with existing legacy object-oriented software system when aspect-oriented becomes standard practice and how to introduce aspects to an existing legacy object-oriented software system are emerged. The traditional object-oriented programming technique has known limitations, which it could not modularize the implementation of crosscutting concerns in existing software systems. Thus, applying AOP technique to transform the legacy object-oriented software system is one way to obtain the benefits of aspect-oriented programming technique.

In this dissertation, we explore the approach to resolve two obstacles in the transformation of object-oriented (OO) program to aspect-oriented (AO) program with static program analysis techniques. First, we propose a new heuristic algorithm for optimizing the combination of input metrics for clustering based aspect mining. Aspect mining is the first phrase in the aspect-oriented refactoring (AOR). Aspect mining aims at detecting the code which is likely to implement a crosscutting concern. Aspect refactoring is the second phrase in the AOR, while it is a way to remove the code implementation of crosscutting concerns from an object-oriented program into aspects. Second, After aspect refactoring, the pointcuts is created with pointcut abstraction rules. The pointcuts are always implemented as name-based or enumeration pointcuts, however, these pointcuts are known as fragility against program evolution. We propose a framework Nataly to solve the fragile pointcut problem by (1) automatic inferring the intention properties from the join points matched by the given name-based pointcuts and (2) generating a method to check whether the given join point satisfies the properties or not. We also give a deep discussion of aspect interference problem in the AOR. In this discussion, we propose an idea to check the aspect interference by using static control flow analysis.

The result of experiments revealed that the metric selection approach increased the accuracy of aspect mining in the aspect mining. The fragile pointcut problems are also alleviated, because the pointcuts which are generated automatically by our framework are more robust than the name-based or enumeration counterpart.

Acknowledgments

Foremost, the author wishes to express his sincere gratitude to his principal advisor Associate Professor Masato Suzuki of Japan Advanced Institute of Science and Technology for his constant encouragement, advices, kindly guidance and support.

The author would like to extend his gratitude to the rest of his thesis committee: Professor Kokichi Futatsugi, Associate Professor Toshiaki Aoki and Professor Kazuhiro Ogata of Japan Advanced Institute of Science and Technology, and Professor Hidehiko Masuhara of Tokyo Institute of Technology for their valuable reviews, advices and suggestions.

Last but not least, the author is grateful to assistant professor Tomoyuki Aotani of Tokyo Institute of Technology for helpful discussions, suggestions and continuous encouragements.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In software engineering, humans cannot concentrate on many concerns or subjects simultaneously. Instead, they abstract from most of the concerns so that they can concentrate on a particular concern in order to cope with it effectively. To concentrate on such particular concerns, developers break, or decompose complex problems into several sub-problems. This is a basic idea of the technique of separation of concerns which are proposed by Dijkstra [2] and Parnas [1].

A concern is used to organize and decompose a software system into manageable and comprehensible parts [3]. Separation of concerns is the ability to keep every concern in its own unit of modularity. Modularity gives a great contribution to software development. It lets humans are able to solve an issue of the software system at a time. In this case, human can make things easier for themselves. Units of modularity include functions, procedures, modules, method, classes and packages and so on. Modularity is a way of achieving separation of concerns. Effective separation of concerns makes a system easier to understand, change and debug [4].

Object-oriented programming provides a way of modularizing concerns. Classes in object-oriented model perform a single specific function. However, not all of a software system's code can be separated. Code related to a concern that cannot be modularized usually results in code scattering and code tangling. Code scattering represents small code fragments spread over several different units of modularity. Code tangling represents intertwined mixture of code fragments in an available unit of modularity. Concerns that suffer from such problem are known in the literature as crosscutting concerns, which its code cuts across several units of modularity. Scattering and tangling increase the difficulty in reading, understanding, maintaining and reusing code.

In many cases, we find that many parts of software systems have code fragments for different exception handling strategies, synchronization policies, distribution strategies, different supports for monitoring, logging optimisation, debugging, authorization and other such tasks. Solving such crosscutting concerns can be quite difficult when using object-orient programming technique. For instance, dealing with the implementation of concerns that generally spread over several different modules of the system; such codes even aren't related related to the function implemented by the module. For that, we need more powerful composition mechanisms. Aspect Oriented Programming (AOP) is a novel technique that improves on object-oriented programming (OOP) in this regard.

AOP aims at solving the problems of scattering and tangling, by modularizing crosscutting concerns in a unique unit, called aspect. Aspect encapsulates behaviors which affect multiple unit of modularity into reusable modules. For example, a software system can be developed according to its main functional requirement, which the possibility for non-functional requirement (such as monitor and persistence) can be defined in a separate aspect.

1.2 Problem Statement

Aspect-oriented programming provides explicit constructs to develop software systems whose concerns are better modularized because they are no longer tangled together and they are clearly separated from farraginous construct into a standalone unit. In practice, AspectJ [5] is the most prevalent one in the mainstream aspect-oriented programming languages. It is an extension of Java [6] that includes mechanisms to support the concepts of AOP. AspectJ introduce three elements: join point, pointcut and advice. A join point is a point in the execution of a program where will be inserted the advice code. A pointcut describes a set of join points. An advice consists of pointcut and code body. Advice is executed at the join point which is matched by its pointcut.

To extend the benefits of aspect-oriented technique to a large number of legacy object-oriented programs, a refactoring approach is proposed by many researches [7, 8] that refactoring the code of crosscutting concerns into aspects. The aspect-oriented refactoring consists of two important phases: aspect mining [9, 10] and aspect refactoring [11, 12, 13]. In aspect mining the problem is analyzing the source code of legacy program for detecting the code slides which are likely to implement the crosscutting concerns. The aspect refactoring focuses on how to remove these crosscutting concerns from the source code of object-oriented program and move them into aspects.

However, there are two significant problems obstacle the refactoring of object-oriented program into an expected aspect-oriented program.

First of all, there is little or no knowledge about how to choose the helpful metrics for aspect mining. Method level aspect mining is the activity of classifying methods that belong to the crosscutting concerns, in a given object-oriented program. A number of researches [14, 15, 16, 17] intend to achieve the aspect mining by using clustering techniques which can classify the similar concerns into the same groups. Each group is called a cluster and it consists of methods which are similar [18]. A clustering algorithm classifies the methods in terms of metrics [19], which are introduced for measuring how program constructs (class, method, field, etc.) are used in the design and development process. However, it is difficult to decide whether the given metrics are suitable or not for each clustering algorithm.

Second, when the code slice of crosscutting concern is refactored into aspect. For example, in AspectJ [5], the pointcut always depends on specific names to match the join points (e.g., enumeration of join points). In this case a fragile pointcut problem arises. A pointcut in a version of the program is said to be fragile if, after the program evolves, (1) it does not match all the expected join points or (2) captures unexpected join points. The refactored pointcuts are particularly fragile when they are written in an enumerative form or specific name-based pattern, whereas an intentional pointcut is expected to be more robust.

1.3 Approach

In this dissertation, the problems of refactoring a legacy object-oriented program to an aspect-oriented program have been addressed by dividing it into three phases.

In the aspect mining phase, we propose a heuristic metric selection algorithm that optimizes the combination of metrics for a given clustering-based aspect mining method. Metric is obtained by static analyzing the structure of object-oriented program. For a given combination of metrics, the selection algorithm selects one metric from the combination into an input metric set and evaluates the quality of such input metric set in an iterative process. In each step a metric is selected from the given combination and moved into the input metric set. How to choose the metric from the given combination is decided by an evaluation algorithm for each step. The selection algorithm outputs the input metric set, which obtains the best quality, for the clustering algorithms.

Next, after the aspect refactoring phase, we propose a framework that translates name-based pointcuts to analysis-based pointcuts automatically. Analysis-based pointcuts match join points with an intention pattern which is the core concept of analysis-based pointcuts that abstracted from the relationship graphs by static analysis. Relationship graphs describe the relationships (e.g., method call, field declaration) between the program elements (e.g., classes, methods). The common/stable vertexes within the relationship graphs are kept in the process of abstraction of intention pattern. Such intention patterns are more stable than the immediate name-based one, so that the analysis-based pointcuts give contribution to the robustness of aspect against program evolution.

1.4 Discussion and Future work

In the aspect refactoring phase, the detected codes of crosscutting concerns are moved into aspects. Such aspects codes inject the crosscutting concerns in the program when the aspect weaver weaves the program. We observe that when such aspects are composed the interference between them are potentially dangerous and can result in erroneous behavior of the generating aspect-oriented program. Without dealing with such interference, the generating aspect-oriented program probably is broken after aspect refactoring. Therefore, in this dissertation, we give a further discussion about this problem and propose an idea to cover this problem. We propose an idea for checking the aspect interference by using control flow analysis in the future. In our idea, first, we extract a constraint property from the OO progress. Second, we extract a model from the generating AO program. We change such problem to a satisfiability problem. We expect to use Satisfiability Modulo Theories (SMT) solver to check whether the model satisfy the property. If it does not satisfy, it indicates that aspect interference potentially arises.

1.5 Dissertation Outline

This dissertation consists, apart from the introduction, seven chapters.

Chapter 2 presents the general research background behind our approach for transforming object-oriented program to aspect-oriented program. In particular, the static program analysis is presented as the basic method for our work. In addition, the aspect mining and aspect refactoring are two significant phases in the transformation process.

Chapter 3 starts to present the metric selection for aspect mining. Why need to select the suitable metrics for the clustering-based aspect mining is explained first. Next, a formalized metric representation and two primary algorithms for evaluating the input metric and find the optimized one are described in detail in this chapter.

Chapter 4 presents a framework Nataly which translates the name-based pointcuts (or enumeration pointcut) to analysis-based pointcuts automatically . This chapter also discusses the fragile pointcut problem after refactoring the object-oriented code to aspects.

Chapter 5 illustrates details of case studies first, and assesses the performance and effectiveness of our approaches through case studies and experiments.

Chapter 6 give a further discussion about the aspect interference awareness. We describe two example to illustrate such problem. We also propose an idea to show a smart way to resolve aspect interference problem in the future work.

Chapter 7 summarize some related works to our research.

Chapter 8 re-states the major contributions of this dissertation and outlines directions of future work.

Chapter 2

Setting the Scene

In this chapter, we will present background material for better understanding this dissertation.

Section 2.1 introduce the research background, which covers work in different research domains that our dissertation draws on, including crosscutting concerns, aspect-oriented programming, a mainstream aspect language-AspectJ, static program analysis and the fragile pointcut problem of aspect-oriented program.

Section 2.2 and section 2.3 introduce two important phases in the transformation of object-oriented program in aspect-oriented program: aspect mining and aspect refactoring. The first phase is to identify code which is likely to implement a crosscutting concern from the object-oriented program. The second phrase is to refactor the codes of crosscutting concerns into aspects without changing their behavior.

2.1 Research Background

2.1.1 Static Program Analysis

Static program analysis is performed on source code or object code in most cases, and is performed without actually executing programs. It predicts dynamically generated behaviors of a program when executing them [21]. Typical examples would be compute where values derive from and may flow to, what possible values that expression may be evaluated to, and what values may reach a certain program point of interest.

Static program analysis is used to optimize code generally [22]. A growing number of static analyses have been used in the verification of properties for safety critical software as well as discovery of bugs in potentially vulnerable code [23].

The analysis provides approximate properties of the programs, which are usually divided into three classes according to their analysis nature:

- **Over approximation** captures the entire behavior of a program. It estimates the program behaviors that may happen along all the execution paths.
- **Under approximation** captures a subset of all possible behaviors of a program. It estimates the program behavior that should happen along all execution paths.
- **Undecidable approximation** cannot decide whether the approximation behavior belongs to the program or not. Its result usually cannot give meaning full information.

In order to get a satisfactory analysis result, the types of analysis technique will be adopted according to the nature of the properties.

It is important that the program analysis should be semantics based, which means that the information obtained from the analysis can be proved to be correct with respect to the semantics of a programming language. There are various types of analysis techniques for answering analysis questions on programs with different language constructs, the mainstream techniques are control flow analysis, data flow analysis, abstract interpretation and type system [21].

Model checking [24] is complementary to static analysis techniques. It can test whether the given model meets certain specifications by performing exhaustive exploration of the possible states in a software system. Model checking is a powerful technique for precise verification of software and hardware. However, it suffers a state explosion problem, and becomes intractable for program with large state spaces and undecidable for infinite one.

In this dissertation, we use static analysis to understand the behavior of the program, the relationship between the different program elements and find the potentially malicious code. In Chapter 3, we propose a static analysis approach to calculate the quality metrics for the software structure. In Chapter 4, we use a control flow analysis to obtain the control flow graph of an object-oriented program and an interprocedural control flow graph of an AspectJ program, respectively. In Chapter 5, the static analysis is used to generate the relationship graph (e.g., call graphs) and properties which support for matching join points.

2.1.2 Crosscutting Concerns

Some concerns often interact with each other in such a way that they cannot be encapsulated properly within object-oriented constructs. These interactions lead to code tangling and code scattering. Code tangling represents when the elements of code for two concerns are in the same unit and cannot be dissociated, while code scattering represents when a concern involves code spread across several units. The concerns which are related in such a way that they imply code scattering or code tangling are said to crosscut each other. A concern that crosscuts the main functional units is a crosscutting concern.

For example, in a software system that provides its users an e-mail service and a file repository service, the two sets of objects that provide theses service will have an authentication concern in common. It is a typical example of code-scattering. If monitoring certain activities in a certain context is required, statements will be added in between the functional code. It is a typical example of code-tangling. More generally, tracing and logging are typical examples of concerns that almost always crosscut the main functional concern: the concern of tracing the behavior and a component is different to the main concern that is implemented by this particular component.

The code of example is shown in Figure 2.1. A Point includes x and y coordinates. A Line is defined as two points p1 and p2. Methods setX and setY of the Point involve two distinct actions. One is updating coordinates in their target object. The other is triggering the redrawing of the display. Updating coordinates x or y of the objet Point is clearly corresponds to methods setX and setY, respectively. Similarly, methods setP1 and setP2 of the Line involve the same actions, Updating p1 or p2 of object Line is clearly correspond to the methods setP1 and setP2, respectively. However, the concern of updating the display has to be handled after execution of setX, setY, setP1 or setP2.

```
Class Point{
  void setX(int x){
    this.x=x;
    Canvas.repaint();
  }
  void setY(int y){
    this.y=y;
    Canvase.repaint();
  }
  //...
}
Class Line{
  void setP1(Point p1){
    this.p1=p2;
    canvase.repait();
  }
  void setP2(Point p2){
    this.p2=p2;
    canvase.repaint();
  }
  //...
}
```

Figure 2.1: Outline of Class Point and Line

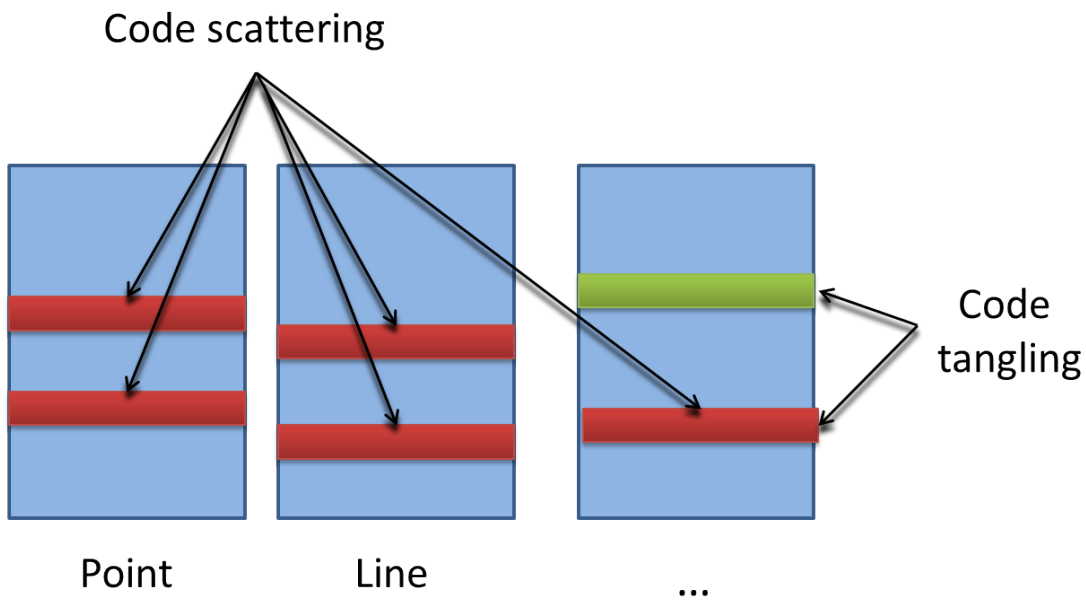

Figure 2.2: Code Scattering and Tangling

Therefore, the display repainting concern crosscuts four methods within two classes, and this concern has to be implemented in several classes.

In Figure 2.2, the red part represents the display repainting crosscutting concerns, and green part represents a monitoring crosscutting concern. These extra concerns not only tangle with basic figure operation functionality but also scatter across numerous other methods in different classes. Improper handling of these concerns leads to loss of modularity and thus decreases the comprehensibility and maintainability of software systems.

2.1.3 Aspect-Oriented Programming

Aspect-oriented programming (AOP) [25] emerged and acted as another effective mechanism for tackling the issue of crosscutting concerns. AOP aims at enable the clean modularization of crosscutting concerns. Crosscutting concerns which have been introduced in the previous section often implements many non-functional requirement and respect in code scattering and code tangling. For instance, security, logging, persistence and transaction control.

Kiczales et al. [25] propose a new programming technique they dubbed aspect-oriented as a solution to the crosscutting effect and associated problems. They use the term aspect to refer to concerns or design decisions difficult or impossible to capture cleanly, no matter which programming model is used. AOP enables the representation in its own unit of modularity of code relative to concerns which crosscut several units. A novel construct which encapsulate the crosscutting concern in a standalone unit of modularity, called aspect. The various aspects are then composed through a process dubbed weaving [26, 25], which produces the application through composition of all the intended aspects.

Aspects are composed by weaving aspect code with a base code. This process works by inlining the instructions from the aspect code into the base code. Although, this

process will produce a tangled version, however, it is tangled does not impact negatively the development process, since it can be regenerated every time when a new version is required. The inlining analogy used to describe weaving may suggest the processing and generating of source code, however the earliest aspect-oriented compilers started to perform this inlining at the binary level, very soon in the technology's developing process. This made it clear that weaving could be regarded as more phase in the execution of a compiler, and that is how it is presently regarded [27]. There is nothing in the aspect-oriented model stating that the target of the weaving process must be a source or binary code, or that the weaving process must be static, dynamic, something in between (e.g., load-time), or supported at the virtual machine level [28].

AOP is meant to complement other models, not to replace them. It is important to realize the relative nature of a concept: a given concern may be crosscutting using one model but may be a cleanly encapsulated one using another. For instance, the procedural model decomposes systems according to algorithms, and therefore data is scattered across multiple units of modularity. In object-oriented programming, data and functionalities comprise the primary decomposition of the system and therefore data is cleanly modularized. Other, non-functional, concerns stand out as crosscutting. This is the reason that the aspects complement OO program tends to be non-functional.

Often, the various aspects relate to different abstraction domains. Recognizing this fact, the earliest proposals presented by Kiczales et al. include aspects coded in different, sometimes concern-specific, languages. This is consistent with the idea of maximizing intentionality. It is the task of the aspect weaver to compose the various aspects into a single application. For instance, Lopes developed two concern-specific languages COOL and RIDL [29, 30] designed to express policies of synchronization in multi-threaded environments and the management of access to remote objects throughout networks. Although Domain Specific Languages (DSLs) alone yields maximum intentionality to model specific domains, use of DSLs is also very limiting, since it would require a new DSL as well as its own specific aspect weaver for each new domain, and possibly for each target application language or platform. These problems are not present in the AspectJ language, which is general-purpose.

2.1.4 AspectJ

AspectJ is a mainstream aspect-oriented tool [5, 31, 32], which is a backwards-compatible extension of Java. Various aspect-oriented tools have later appeared [33], but the majority is strongly influenced by the design of AspectJ. Currently, AspectJ becomes a standard for AOP in terms of language design. For this reason, we propose the problems and the approaches are all based on Java and AspectJ, in addition, we use the terms AOP and AspectJ interchangeably throughout this dissertation.

AspectJ was designed to be independent of any specific implementation, so that widely different approaches could be taken to support its mechanisms. For instance, an AspectJ weaver can take source code as input, or binary code, and its output can be source code (in which case will have to be compiled again by another compiler), or will directly output binary code. Depending the specific implementations, the weaving process can take place at compile-time, run-time or class load-time, or be performed by the virtual machine itself. The first implementation developed at Xerox Palo Alto Research (PARC) included a weaver that accepted the source code of the target classes as input and directly

generated bytecode-compatible binaries. Later, a second implementation was developed that supports bytecode-weaving, so both input and output come in binary form. AspectJ now is a part of IBM's open source Eclipse project.

AspectJ uses regular Java statements to write the advice, however it defines a lot of specific constructs for encapsulating aspects and writing pointcuts. For example, some member functions or data that are related in functionality may be part of different classes or nested within different functions. Nevertheless, the aspect construct can still encapsulate them. In order to understand this dissertation well, we introduce some details of AspectJ.

Join Point: Join point is a novel concept that makes oblivious quantification [34] possible. A join point is any identifiable execution point in software system [32]. For example, a call to a method could be a join point, an access to a field. In addition, a join point shadow is a location in which advice-execution instruction can be inserted. The following execution events are all examples of join points:

- The call to a method
- The execution to a method
- The execution of an object's creation logic
- The invocation of an object's creation logic
- The access to a field for reading
- The access to a field for writing
- The execution of an exception handler
- The loading of a class
- The Initialization of an object in a constructor
- The pre-initialization of an object in a constructor

Pointcuts: AspectJ includes a novel construct, the pointcut designator, which are written as logical expression defining which of these join points need to be identified. The join points are captured by a pointcut designator correspond to non-contiguous places in the program's source code. Pointcut expression can include wildcards and specify an open-ended set of join points. Table 2.1 shows a few legal examples of AspectJ's pointcut protocol.

Pointcut in AspectJ include `call`, `execution`, `initialization`, `handler`, `get`, `set`, `this`, `args`, `target`, `cflow`, `cflowbelow`, `within`, `withincode`, `if`, `preinitialization` and `adviceexecution`. A call pointcut invokes a method, and a handler pointcut captures the execution of an exception handler in an application. A typical format for pointcut is:

$$pointcut \quad pointcutName([parameters]) : designator(joinpoint)$$

`pointcutName` is the name of pointcut, and is used to handle actions. `designator` decides when join point will match. A `designator` is used to expose the object to advice, or to a narrow pointcut selection. Designators include `this`, `target`, and `args`. The designators

Signature pattern	Description
call (void Foo.m(int))	A call to the method with signature void m(int) in class Foo
call (void m*(int))	Calls to void methods whose name starts with an 'm' and which have a single parameter of type int
call (* m*(..))	Calls to methods of any type whose name starts with 'm' and with any parameters
execution (! public Foo.new(..))	The execution of any non public consruction of Foo
initialization (Foo.new(int))	The initialisation of any Foo object that is constructed with construction Foo(int)
staticinitialization (Foo)	When the type Foo is initialised, after its class being loaded by the virtual machine
get (int Point.x)	When an access for reading the integer field x in the Point class takes place
set (! private * Point.*)	Access for writing the value of any non-private field in the Point class
handler (IOException+)	When an IOException or one of its subtypes is handled with a catch block

Table 2.1: Examples of pointcuts in AspectJ

of `cflow` and `cflowbelow` match join points within a given program flow, whereas, within and `withincode` match classes and methods. A pointcut can be combined with three logical operators and may be combined using set operators such as:
p_1 && p_2 : it represents all join points in both p_1 and p_2
p_1 || p_2 : it represents any join point in p_1 or p_2
p_1 && !p_2: it represents any join point in p_1 and should not in p_2

Advice: The constructs that specify how the behavior of the base code is to be affected are the advice. Advice is nameless method-like blocks associated with a given pointcut that execute implicitly, whenever one of the join points is reached. Note that although advices resemble methods in some way, they do not have call semantics: advices are never called explicitly, and therefore do not need a name.

Advice can define and use their own temporary variables the same way as methods, and can use and modify whatever values are exposed by the advices signature. The same way methods can use and modify values received as parameters. Advice in AspectJ supports before, after and around:

Before Advice The advice executes immediately before the join point is reached and it cannot prevent the join point from executing.

After Advice The advice executes immediately after a join point has finished, and therefore does not prevent it from executing. After advices can be classified into three different advices. The first one is unqualified after advice which executes regardless of the result of the join point. The second one is after returning advice which executes after the successful execution of join points. The third one is after throwing advice which is executed only when the advised join point throws an exception.

Around Advice The original join point is executed only if the advice explicitly calls it, using the proceed keyword. The around advice should declare a return type, which must conform to the type of the join point which triggered it.

2.1.5 Fragile Pointcut Problem

In this section, we explain the fragile pointcut problem, provide an analysis of the possible cause of the fragility of pointcut definitions, and illustrate each of them through examples.

Definitions

The fragile pointcut problem[36] is similar to the fragile base class problem in object-oriented programming. Software engineers cannot determine whether the base class change is safe only by inspecting its methods independently when in OO development. In addition, they also need to inspect the methods of subclasses. Translating the problem to aspects, with the purpose of determining whether the base program change is safe, developers have to inspect possible influences in the join point shadows which captured by the particular pointcuts in the program. According to [35, 36], pointcuts are fragile because when we change the original program, the semantics of pointcuts may change 'silently', even though the point definition itself is not changed. To match the changed set of join points, we need to change the semantic of pointcut which captures such join points. Kellens et al. [37] give a definition of the fragile pointcut problem:

Definition 1 *Fragile Pointcut Problem occurs in aspect-oriented software when pointcuts unintentionally capture or miss particular join points as a consequence of their fragility with respect to seemingly safe modifications to the base program.*

In an aspect-oriented program, no one can tell whether a change to the base code is safe simply by examining the base program is isolation. Any change to the structure of behavior of the base program would impact the join points which are captured by the pointcut definitions, because these pointcuts capture join points based on program structural or behavioural property shared by those join points. For name-based pointcuts, they rely on the names of methods, classes, and fields are typical examples. Suppose for a simple example, a name-based pointcut matches join point which is the execution of method m. If the name of the method is changed from m to n, then the pointcut has to change. If in the evolution version of the program, source code entities are changed accidentally leads to the capture of a join point related to these source code entities, Kellens et al. [37] describe that it is an unintended join point capture. Conversely, when the base program is changed is in such a way that one of the join points which was originally captured by the pointcut is no longer captured, even though it was still supposed to be captured, they describe that it is an accidental join point miss.

Examples

In order to understand the fragile pointcut problem clearly, we explain about two examples of the fragile pointcut problem, namely accidental join point captures and misses. We use the Figure Editor application as a benchmark.

The Figure Editor application has a canvas and provides points and lines as the primitive graphical elements to draw a figure on the canvas. It repaints the canvas if the user adds or moves points and/or lines on the canvas.

```
1  class Point implements FigureElement{
2    int x,y;
3    void setX(int x){
4      this.x=x;
5    }
6    void setY(int y){
7      this.y=y;
8    }
9    void moveBy(int dx, int dy){
10     x+=dx;
11     y+=dy;
12   }
13   //...
14 }
15 class Line implements FigureElement{
16   Point p1, p2;
17   void moveBy(int dx, int dy){
18     p1.moveBy(dx, dy);
19     p2.moveBy(dx, dy);
20   }
21   //...
22 }
```

Figure 2.3: Figure editor example

Fig. 2.3 shows the classes `Point` and `Line`, respectively. A `Point` is defined by `x` and `y` coordinates and provides the setters (`setX` and `setY`). The following method `moveBy` changes the value of `x` and `y`. A `Line` is defined by two points `p1` and `p2`. It provides method `moveBy` changes the value of `p1` and `p2`.

Fig. 2.4 shows the implementation of the repaint function in aspect `CanvasRepaintAspect`. A traditional pointcut descriptor in this aspect expresses the points in the execution flow from the methods `setX`, `setY`, `Point.moveBy` and `Line.moveBy`. It specifies execution of the methods whose name begin with the text "move" and "set" in class `Point` and `Line`. The piece of advice code associated with this pointcut repaints canvas.

Example 1: Accidental Missing

Fig. 2.5 shows that the programmer adds a new field `color` with its associated method `changeColor` in class `Line` and `Point`. The new method represents an operation that changes the color property of a line and a point.

The fragility of traditional pointcuts in this scenario comes from accidental misses. The pointcut `figureChange` cannot capture new method `changeColor`, because the name of method `changeColor` does not begin with the text "move" or "set". In this case, when the color property of a line or a point is changed, the program cannot repaint the canvas.

Example 2: Accidental Capturing

```
aspect CanvasRepaintAspect {
  pointcut figureChange():
     execution(* FigureElement+.move*(..))||
     execution(* FigureElement+.set*(..)) ;

  after(): figureChange(){
     Canvas.repaint();
  }
  //...
}
```

Figure 2.4: Outline of the aspect CanvasRepaintAspect with traditional pointcut

```
Class Point implements FigureElement{
//...
  Color c;
  void changeColor(Color c){
    this.c=c;
  }
}
Class Line implements FigureElement{
//...
  Color c;
  void changeColor(Color c){
    this.c=c;
    p1.changeColor(c);
    p2.changeColor(c);
  }
}
```

Figure 2.5: Programmer adds new a method changeColor in class Point and Line

Fig. 2.6 shows the programmer adds a new field `date` with its associated method `setDate` in class `Point` and `Line`. New method `setDate` records the last time when a figure element is saved in a persistent store.

The fragility of name-based pointcuts in this scenario comes from accidental captures. The pointcut `figureChange` captures the new method `setDate` because its name begins with the text "set". After the program executes the method `setDate`, it will repaint the canvas. However the method `setData` does not change any figure element's visual properties.

The problem occurs because the pointcut `change` does not match correct join points; it accidentally matches the execution of new method `setDate` in the first case, and misses the execution of method `moveBy` in the second case. We call the first problem accidental join point capture and the second problem accidental join point miss, respectively, in this dissertation. In order to solve these problems, pointcut languages should be improved in order to loosen the coupling between aspect and base code's structure.

```
Class Point implements FigureElement{
//...
  Date d;
  void setDate(Date d){
    this.d=d;
  }
}
Class Line implements FigureElement{
//...
  Date d;
  void setDate(Date d){
    this.d=d;
  }
}
```

Figure 2.6: Programmer adds a new method setDate in class Point and Line

2.2 Aspect Mining

Manually applying aspect-oriented technique to a legacy system is difficult and error prone process. Because of the large size of such software systems, the complexity of the implementation, the lack of documentation and knowledge about the system, there is a requirement for developing a technique that can help software engineers in locating or documenting the crosscutting concerns in legacy systems.

Aspect Mining is the first phase to implement the transforming of the legacy object-oriented program to aspect-oriented program. Aspect mining is the activity of detecting the implementation of crosscutting concerns from an object-oriented program. In this section, we introduce the activity of aspect mining and have a survey of existing aspect mining techniques.

Recurring patterns: Breu et al. [38] propose a Dynamic Aspect Mining Tool named DynAmit. Their tool DynAmit analyses the traces of a program which reflecting the run-time behavior of a system, for searching the recurring execution patterns. In order to implement this, they propose the notion of execution relations between method invocations. In the following is an example of an event trace:

```
A(){
  B(){
       C(){}
       D(){}
     }
   }
E(){}
```

There are four different execution relations are discussed in DynAmit: outside-before (A is called before E), outside-after (E is called after A), inside-first (C is the first call method in B) and inside-last (D is the last call method in B). The mining algorithm detects crosscutting concerns based on recurring patterns of method invocations. We assume that an execution relation is detected as a crosscutting concern if it recurs uniformly. In

addition, the recurring relations should appear in different calling contexts. Although this approach is inherently dynamic, Brue et al. use control flow graphs to calculate the call relations in their experiment for many times. To remove ambiguities and improve the result, their approach uses static type information to supplement the dynamic information [39].

Formal concept analysis: Formal concept analysis (FCA) [40] is a branch of lattice theory. Given a set of objects and their attributes, such approach creates concepts of a group of objects which have common attributes. FCA approach organizes such concepts into a lattice on the basis of the partial order associated with attribute set inclusion.

Dynamo [9] is one of the aspect mining tools which apply FCA. When using it to analyze a software system, an instrumented version of the software system is executed on a number of use cases, manually derived from the software documentation. The output of this execution is a number of execution traces. In the FCA algorithm, the use cases are the objects, while the methods which get invoked during the execution of a use case are the attributes. As a result, all concepts are selected which contain traces from exactly one use case. If the specific attributes of the concept belong to more than one class and different methods which belong to the same class, are specific to more than one use case specific concept. As a result, these concepts are considered as possible aspects.

Natural language processing: Natural language processing is under the assumption that the implementations of crosscutting concerns are often on the basis of the rigorous naming and coding convention, which is considered as an indicator for possible aspects. Shepherd et al. [41] use lexical chaining [54] to find the implementation of crosscutting concerns. Lexical chaining uses a given collection of words as input, and output chains of words which are semantically strongly related. A semantic distance measure between each combination of words is always used here to create the chains.

Detecting unique methods: This mining technique is on the assumption that crosscutting concerns are often implemented in an idiomatic way [42, 43]. These idioms are regarded as symptoms of crosscutting concerns. An example of such idiom is the implementation of a crosscutting concern by means of a single entity in the software system which is called from numerous places in the code. A new concept of unique methods is defined as "A method without a return value which implements a message implemented by no other method" in [42]. The unique methods should be filtered first, and then the possible aspects are inspected by the software engineer manually.

Clustering-based aspect mining: Clustering is a technique that classifies objects into different groups. Each group is called a cluster and it consists of similar objects. The objects which are in different groups are dissimilar to each other [44]. Applying the clustering technique, the crosscutting concerns would be classified into similar groups and other general concerns are divided into other groups. We introduce two types of clustering algorithms here, which are applied in this dissertation.

- **Hierarchical clustering algorithm (HC)** The hierarchical clustering organizes objects into a tree [45]. Similar leaves are within the same sub-trees. Leaves also represent genes in HC. The length of the paths between leaves represents the distance between them. The hierarchical approach first often finds the similarity or dissimilarity between each pair of input data. Second, it groups the input data into a binary, hierarchical tree. Finally it determines where to cut the hierarchical tree into cluster.

- **k-means clustering algorithm (KAM)** The k-means clustering algorithm partitions the objects into k groups, where the distance between the object and its assigned cluster centroid is minimized [14]. Data being grouped in an exclusive way (the clusters do not overlap). However, this algorithm can be negatively affected by a single outlier. The k-means clustering algorithm often needs user to assign the number of clusters in advance. IF the value of k is equal to 2, the data will be classified into two groups (crosscutting concerns group and general elements group).

Fan-in analysis: Fan-in analysis aims at finding methods by computing the fan-in value for each method using static call graph of the software system. The fan-in value is considered as a good indicator of scattering. For example, Marin et al. [46] noticed that a lot of the well-known crosscutting concerns exhibit a high fan-in value. Methods which are called often from different context are possible crosscutting concerns. The fan-in analysis approach calculates the fan-in value for each method, filter accessor and auxiliary methods such as toString(), and the number of considered methods is also limited by the fan-in threshold. Thus, because of the threshold, aspects with a smaller footprint may be ignored by this approach.

Clone detection: For the clone detection techniques, these approaches are based on the assumption that crosscutting concerns result in code duplication. There are two techniques on the basis of such observation for mining the possible aspects. Shepherd et al. [47] propose an approach that detects the possible aspects on the basis of program dependence graphs (PDG). Bruntink et al. [48, 49] propose an approach that detects the possible aspects on the basis of three other clone detection techniques (token-based, AST-based and metrics-based clone detection).

2.3 Aspect Refactoring

Refactoring is defined as the process of changing a program's structure without altering its observable behavior [50]. Refactoring is changing codes in such a way that they correspond to a better design. A procedure reverses the traditional order of design and code by sequence. The refactor code is meant to be better organized and easier to maintain, adapt and extend, while providing the same functionality. Each refactoring describes a disciplined way to modify the code in order to achieve a specific design change, without introducing compiler errors, bugs, or otherwise affecting the application's externally observable behavior [50].

Refactoring can be performed either manually or automatically. The earlier research efforts focused on tools which could automatically perform behavior preserving transformations on the source code. A typical example for a behavior-preserving transformation is the remove class refactoring. It only removes the class, if the class exists and if it is not referenced from other parts of the system. The availability of tool support for code transformations is very desirable to provide safety in its use and to increase productivity.

A traditional refactoring technique in object-oriented software system cannot directly apply to aspect-oriented systems because the behavior is not guaranteed to remain preserved. The relationship between object-oriented refactoring and aspect-oriented refactoring is discussed in [11]. In Additional, popular object-oriented transformations have been re-defined in order to make them aspect-aware in Flower [50]. In this section, we

will introduce a number of novel refactoring techniques which transform the code implementation of crosscutting concerns into aspects.

Catalogs of refactoring crosscutting concerns form object-oriented code into aspect code have been discussed in [8, 12]. The feasibility of the approach has been shown in small case studies, such as the observer design pattern [51]. Behavior preservation conditions for aspect refactoring are also formally specified [8].

Even small changes in the base code could make a point fail in matching all the intended join points. This problem is known as the fragile pointcut problem, which has been introduced in section 2.1.5. Pointcuts are particularly fragile when they are written in an enumerative form, whereas an intentional pointcut is expected to be more robust. In Chapter 5, we discuss the fragility of pointcuts which are created in the refactoring process.

Automated support for refactoring object-oriented code into aspects has been investigated by [52, 53, 7]. An approach derived from program slicing [52] improves a popular object oriented refactoring, method extraction. A generalization of this refactoring has been defined to untangle the crosscutting concerns from the base code and to move them into an aspect. A library contains some predefined crosscutting concerns together with those refactoring that should be applied to change each particular concern into an aspect. Both the crosscutting concern and the refactoring are specified in terms of abstract rules. A human intensive effort is required to manually map abstract roles to concrete program elements. Tool supported refactoring [7] presented an approach to refactoring object-oriented programs, written in Java, into equivalent aspect-oriented programs, written in AspectJ. A simple set of six refactoring has been defined to refactor the OO program to the AO program and they implement a tool called AOP-Migrator tool as an Eclipse plug-in.

Software refactoring aims at making the code easier to understand or maintain on the basis of modifying the internal program structure but does not alter the external behavior. However, in all above aspect refactoring techniques, the interactions between the aspects, which are created in the refactoring process, are not coping with. As a result, the behavior of the aspect-oriented program may be unexpected or incorrect compare with the original one. We discuss the details of this problem in Chapter 4 and propose a novel awareness mechanism to deal with these aspect interactions.

Chapter 3

Reliable Metric Selection for Aspect Mining

In order to transform the legacy object-oriented program to aspect-oriented program, finding the crosscutting concerns is the first issue that requires to be resolved. Aspect mining [9, 10] is proposed to detect program elements in a given program that implement the crosscutting concerns. Clustering algorithm is one way to implement automatic aspect mining. Clustering algorithm always uses software metrics as a vector input. However, there is no existing reliable standard metrics for the clustering-based aspect mining approach. How to decide which metrics are suitable for finding crosscutting concerns with clustering algorithm is difficult for the program.

In this chapter, we first illustrate four scenarios of using clustering-based aspect mining to find crosscutting concerns with different metrics in section 3.1. A definition of formalized metrics which is given in section 3.2. In the last section of this chapter (section 3.3), we propose two algorithms. One is an evaluation algorithm which calculates the quality for each metric. Another one is a heuristic metric selection algorithm which finds an optimized set of metrics for a given program.

3.1 Clustering-based Aspect Mining with different metrics

In this section we introduce the metrics which are frequently used in software quality engineering, and explain how significant for selecting a suitable set of metrics for aspect mining with four concrete scenarios. Each scenario uses a different combination of metrics. We use a figure editor application as an example [55], which draws points and lines on a canvas. We utilize the hierarchical clustering method[56] in these four scenarios.

Fig. 3.1 and Fig. 3.2 show the class Point and Line, respectively. The class `Point` is defined by `x` and `y` coordinates, and provides the getters (`getX` and `getY`) and setters (`setX` and `setY`). The following method `moveBy` changes the value of `x` and `y`. The class `Line` is defined by two points `p1` and `p2`, and provides getters (`getP1` and `getP2`) and setters (`setP1` and `setP2`). The following method `moveBy` changes the value of `p1` and `p2`.

After changing the location of a figure element, the canvas needs to be repainted. A glance at Fig. 3.3, the code of the method `needsRepaint` appears in the six methods, i.e.,

```
class Point implements FigureElement{
  private int x=0, y=0;
  Point(int x, int y){
    super();
    this.x = x;
    this.y = y;
  }
  public int getX(){ return x; }
  public int getY() {return y; }
  public void setX(int x){
    this.x=x;
    Display.needsRepaint();
  }
  public void setY(int y){
    this.y=y;
    Display.needsRepaint();
  }
  public void moveBy(int dx, int dy){
    setX(getX()+ dx);
    setY(getY()+ dy);
    Display.needsRepaint();
  }
}
```

Figure 3.1: Outline of the class `Point`

`setX`, `setY`, `moveBy` in class Point and `setP1`, `setP2` and `moveBy` in class Line . Hence, we consider that the method of `needsRepaint` is scattering, which is one of the characteristic of crosscutting concern in the figure editor application. By using the clustering-based aspect mining technique we intend to classify the methods into two groups: Crosscutting concerns group {`Display.needRepaint`} and General group {other methods}.

3.1.1 Metrics

Before illustrate the scenarios, we briefly introduce the metrics, which are used in the following scenarios.

Fan-in Value (FIV): Marin et al. [46] defined Fan-In metric for a method as the number of distinct method bodies, which can invoke the method. In general, modules with a large fan-in are relatively small and simple, and are usually located at the lower layer of the design structure. In contrast, modules that are large and complex are likely to have a small fan-in. In an experiment presented by [46], one-third of the methods found with high Fan-In value were seeds leading to aspects.

Fin-out Value (FOV): The number of distinct methods that are invoked by a method is called FOV. FOV is usually used to measure coupling between components of software systems.

Henry and Kafura's structure complexity (C_p): C_p is defined as the square of

```
class Line implements FigureElement{
  private Point p1, p2;
  Line(Point p1, Point p2){
    super();
    this.p1 = p1;
    this.p2 = p2;
  }
  public Point getP1(){ return p1; }
  public Point getP2(){ return p2; }
  public void setP1(Point p1){
    this.p1=p1;
    Display.needsRepaint();
  }
  public void setP2(Point p2){
    this.p2=p2;
    Display.needsRepaint();
  }
  public void moveBy(int dx, int dy){
    getP1().moveBy(dx, dy);
    getP2().moveBy(dx, dy);
    Display.needsRepaint();
  }
}
```

Figure 3.2: Outline of the class `Line`

the product of FIV and FOV metrics. Is formally defined as :

$$C_p = (fin - in \times fan - out)^2$$

Method Signature (MSig): A common definition of method signature in software engineering consists of method name, parameter types. Return type and exception throws are not considered generally. If a method's signature does not change frequently the signature pattern can be used in determining similarity between methods [62].

3.1.2 Scenario 1: Selecting Fan-out Value/Fan-in Value as Metric

In this scenario, we select only one metric in the clustering algorithm independently to detect the crosscutting concerns.

We distribute the values of metric FOV in 1-dimensional space in Fig. 3.4. There are total 12 points that are distributed in a same line and it is ambiguous to define the clusters on that line. Therefore, only use FOV as a metric independently cannot define the clusters.

We distribute the values of metric FIV in 1-dimensional space in Fig. 3.5. There are also total 12 points that are distributed in a same line. Fig. 3.5 is different from Fig. 3.4

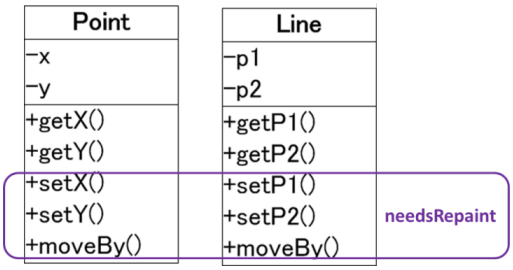

Figure 3.3: Crosscutting concerns in Figure example

in that the clustering algorithm which use FIV takes values in two separate ranges. It can be clearly seen that FIV is more helpful for defining the two clusters than FOV.

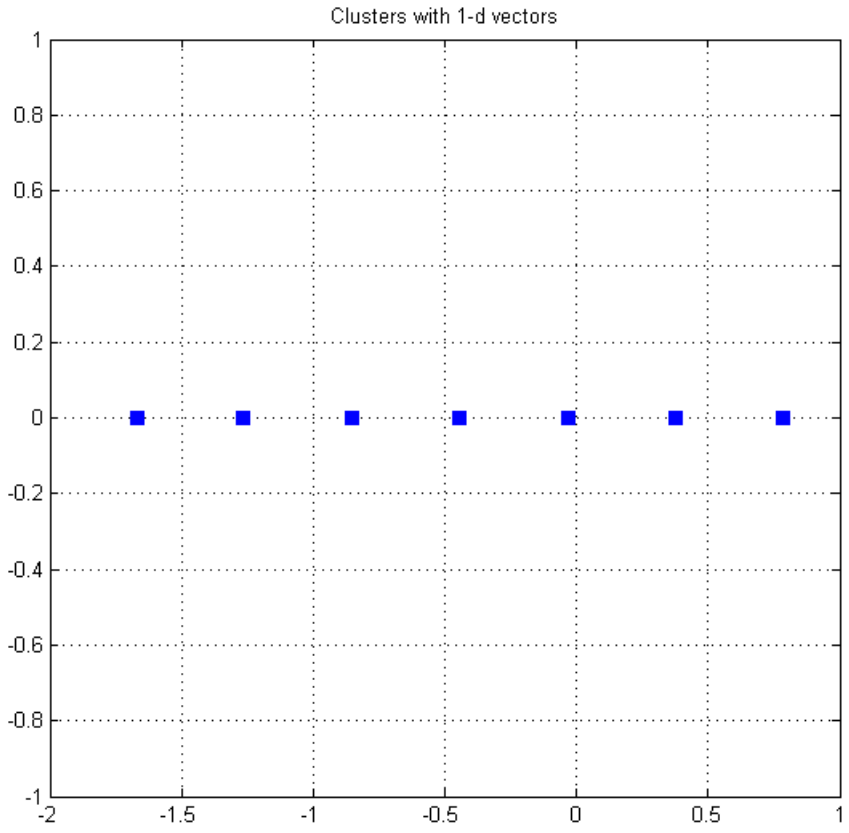

Figure 3.4: Effect of FOV on clustering

3.1.3 Scenario 2: Selecting FIV and FOV as Metrics

In this scenario, we use FIV and FOV as correlated metrics in the clustering algorithm. Fig. 3.6 shows the result of the clustering by using FOV and FIV. There are two clusters with total 12 points that are distributed into two clusters. One cluster in square marker has 1 point that represents the group of crosscutting concerns. Another cluster in circle marker has 11 points (include overlapped points) that represents the group of general

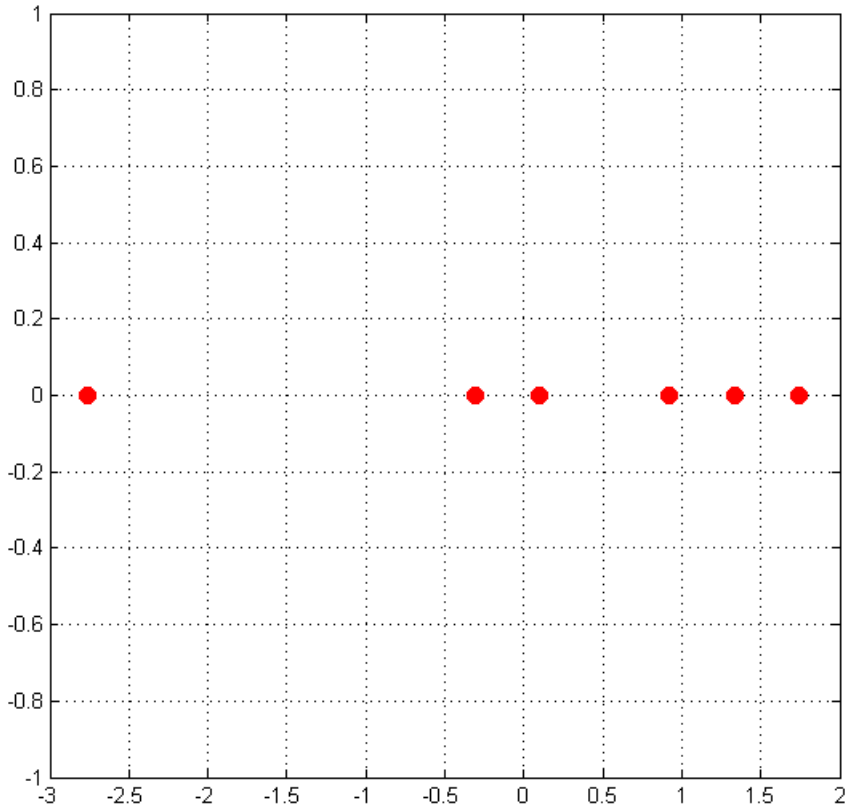

Figure 3.5: Effect of FIV on clustering

program elements. In this case, the crosscutting concerns can be found in the square cluster clearly. We calculated the execution time of this algorithm. It spent 2.1254 second in the experiment.

3.1.4 Scenario 3: Selecting FOV and C_p as Metrics

The developer selects two metrics: FOV and C_p in the clustering algorithm. See from Fig. 3.7, there are two clusters with total 12 points. One cluster in square marker has 3 points that represents the group of crosscutting concerns. Another one in circle marker has 9 points (include overlapped points) that represents the group of general program elements. In this case, the crosscutting concerns can be found in the square cluster. However, the square cluster includes two other methods that are not considered as a crosscutting concern.

3.1.5 Scenario 4: Selecting FOV, FIV and MSig as Metrics

Msig is a metric that denotes method signature, which has been used in other aspect mining research [57]. Fig. 3.8 shows the results of the clustering by using FOV, FIV and MSig. We can see that the methods are classified into two clusters, and the effect is as well as the previous one which uses two metrics {FOV, FIV} in the scenario 2. Thus, we

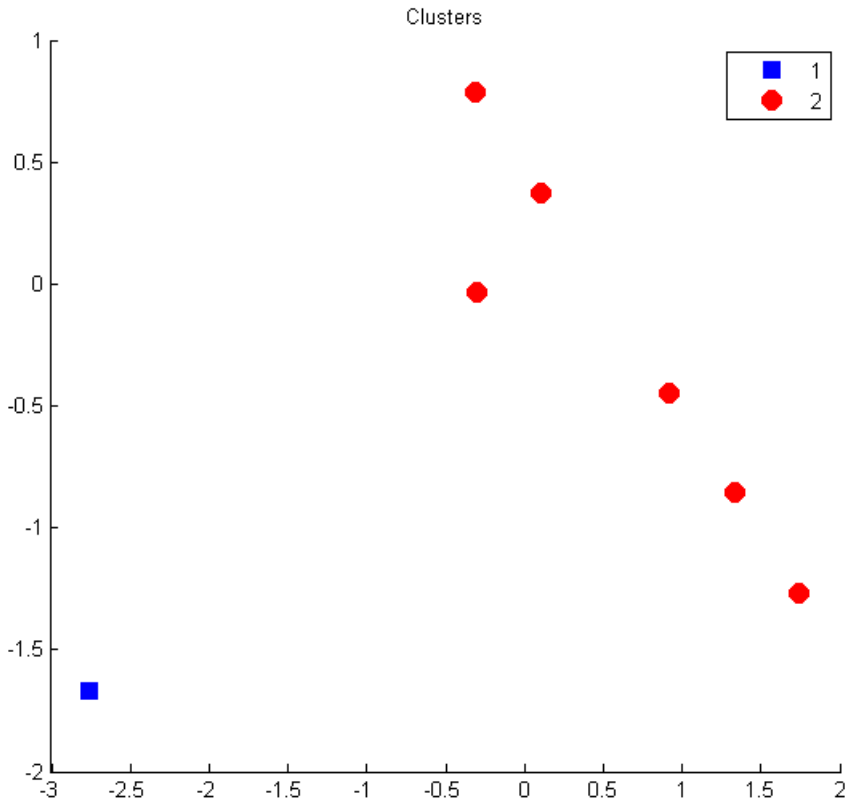

Figure 3.6: Effect of FIV and FOV on clustering

consider that using all three metrics {FOV, FIV, MSig} is unnecessary. We calculated the execution time of this algorithm. It spent 2.2211 second in the experiment. Compare to the scenario2, it spends more time to execute the clustering algorithm.

3.1.6 Summary

In this section, we have described four clustering scenarios that use different combination of metrics. However, some results of clustering are not correct for aspect mining. In the scenario 1 there is no cluster can be found, when programmer only uses FOV as a metric. In scenario 3 the methods can be classified into two groups, nonetheless, the crosscutting concern group does not only include the crosscutting concern, but also includes two other general methods. A further important point is that in this case the aspect mining approach is inefficient and it needs to spend more time to distinguish the un-crosscutting concerns from the crosscutting concerns cluster. Therefore, the metrics {FOV,C_p} are not suitable to classify the crosscutting concerns for the Figure editor application. In scenario 2 and scenario 4, the effects of clustering are same by using the same clustering algorithm. The different point is that one uses two metrics and the other uses three metrics. Thus, selecting metrics {FIV, FOV} reduces the dimensionality of the data while forms well separated clusters. In addition, scenario 2 also saves the execution time of clustering.

The problems are exposed by comparing these four scenarios. The first point is that some metrics can define the cluster independently, but some ones are not. Therefore, in

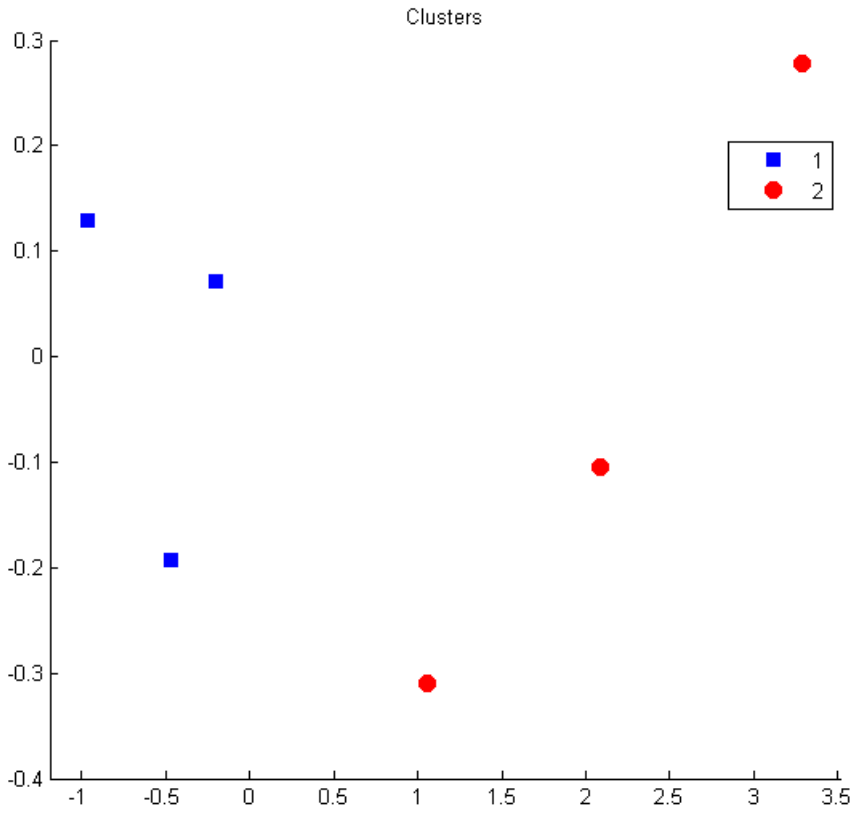

Figure 3.7: Effect of FOV and C_p on clustering

the metrics selection process how to choose a helpful metric as the basic one is the first issue need to be focused. The second point is that some metrics are unnecessary for the clustering algorithm to classify the crosscutting concerns. Deciding which metric is helpful or not manually is difficult. How to optimize the combination of metrics automatically is the other issue. Optimizing the combination of metrics means the accuracy of clustering is as well as or increased even the number of metrics are decreased.

3.2 Formalized Metrics

Based on the observations described in section 3.1, we have defined a formalized metric for guiding programming to select a basic metric at the beginning of the metric selection process.

In this dissertation, classes or packages are considered as a set of module elements and methods are considered as a set of concern elements which will be clustered in a given program. The crosscutting concern is scattered over various modules and wherein at least one of these modules is tangled. Scattering and tangling always appear together. Scattering is caused when single functionality is implemented in multiple modules and tangling is caused when a module is implemented to handle multiple concerns simultaneously. Here, we would like to let the basic metrics focus attention on measuring the scattering, which is one of the characteristic of crosscutting concern.

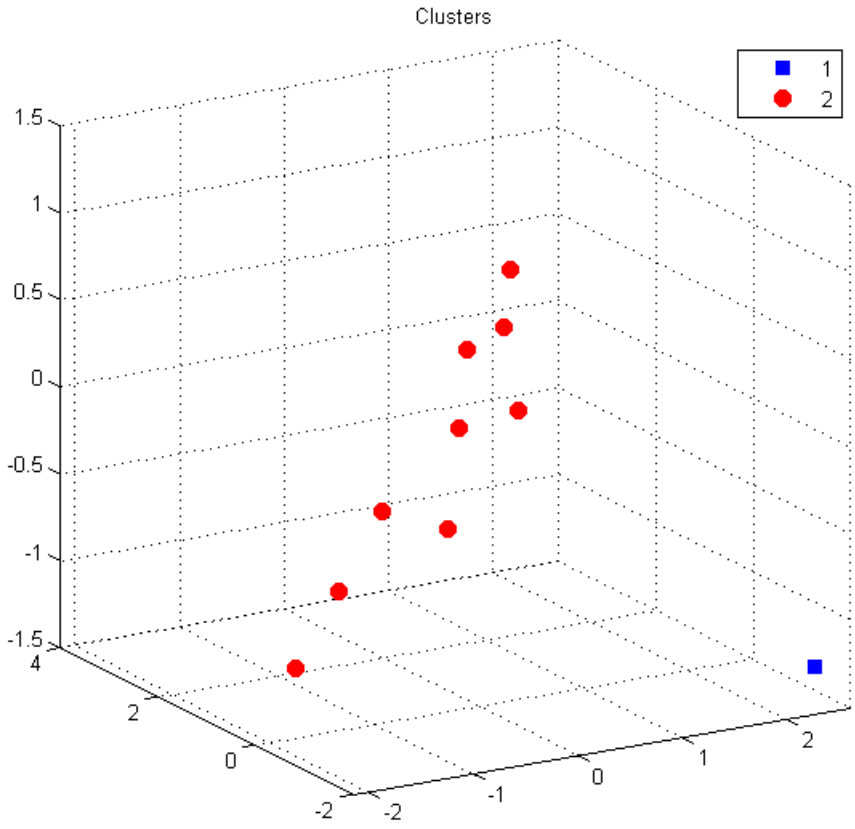

Figure 3.8: Effect of FOV, FIV and MSig on clustering

We define a function f is a trace relationship between a given concern element (method) and a set of module elements (classes).

$$f \subseteq M_{th} \times P(CL_s)$$

where M_{th} is the set of all the methods and $P(CL_s)$ is the power set of classes. We define two formulas $defined_f$ and $called_f$:

$$defined_f(m) = \{C\} \tag{3.1}$$

$$called_f(m) = \{C_1, C_2, \cdots, C_n\} \tag{3.2}$$

Formula (3.1) means class C defines method m. Formula (3.2) means $\forall i \in 1, 2, \ldots, n$. ($\exists k \in M_{th}, defined_f(k) = C_i \wedge$ method k calls method m), where $1 \leq i \leq n$.

Similarly, we give another function g is a trace relationship between a given method and a set of methods.

$$g \subseteq M_{th} \times P(M_{th})$$

where M_{th} is a set of all methods and $P(M_{th})$ is the power set of methods. We define two formulas:

$$defined_g(M) = \{C\} \tag{3.3}$$

$$called_g(m) = \{M_1, M_2, \cdots, M_n\} \tag{3.4}$$

Formula (3.3) means class C defines a set of methods M. Formula (3.4) means $\forall i \in 1, 2, \ldots, n$. M_i calls m, where $1 \leq i \leq n$.

Firstly, we give a formalized definition of scattering as follows: $m \in M_{th}$ is scattered, if $cd\big(called_f(m)\big) > 1 \vee cd\big(defined_g\big(called_g(m)\big)\big) > 1$, where cd refers to cardinality of module elements. It represents a concern element m has a trace relationship with multiple module elements. Alternatively, scattering occurs when in a mapping between concern and module, a concern element is related to multiple module elements.

Secondly, we define the formalized metrics according to the description of the definition of scattering as:

$$E = h_E\big(f_E(m), g_E(m)\big)$$

where E is the metrics, the function f and g are the trace relationship, which have been described in this section, m is the given method. The formalized metric measures concern scattering for a given program. We consider that if a concern belongs to the crosscutting concerns, it should be scattered. We presume that when we use basic metric to measure this concern the result should be different from other general concerns. Based on the ranking of Qe and Te, we can decide to select which input data is suitable for mining.

A simple example of basic metric is as follows:

The metric of affected classes (AC) [57, 58] measures a number of classes which call a given method. The fan-in value (FIV) [46] measures the number of distinct method bodies that call a given method. Considering AC satisfies the formalized metrics of $AC(m) = |f_{AC}(m) \times 1 + |g_{AC}(m) \times 0|$ and FIV satisfies the formalized metrics of $FIV(m) = |f_{FIV}(m)| \times 0 + |g_{FIV}(m)| \times 1$.

3.3 Metric Selection Algorithm

Metric selection is difficult because there are no obvious criteria to guide the algorithm. Consider that if the combination of metrics contains N number of metrics, then the total number of competing candidate subsets to be generated is 2^N. It is a huge number even if the number of metrics is a medium-size. Therefore, this section gives a heuristic algorithm that selects the metrics which have contribution to detect crosscutting concerns.

The selection algorithm is based on two error metrics: Qe and Te. Quantization error (Qe) is the quality error for each input data, which measures the average distance of the sample vectors to the cluster centroids by which they are represented. The higher value is the Qe, the higher is the heterogeneity of the data cluster. Thus, in our approach we select the input data which the Qe value of such input data has a smaller value. Topographic error (Te) is the most simple of the topology preservation measures. The total error is normalized to a range from 0 to 1, where 0 means perfect topology preservation. Thus, our selection algorithm decides to select the input data which has a smaller value of Te.

3.3.1 Evaluation algorithm QeA-SOM

QeA-SOM (Algorithm 1) takes a set of metrics as input, measures the quality for each metric and then returns a ranking list of metrics which are sorted in the ascending order according to their quality. A small value of Qe is more desirable, it represents that input data is better for the mining. The Qe value can be calculated with the existing miming technical such as self-organizing map (SOM) algorithm [59]. SOM is a clustering technique

that converts non-linear statistical relationships between high-dimensional data into two dimensional geometric relationships. The vector space models which are computed by SOM are organized into two-dimensional order. In such two-dimensional order similar models are close to each other in the grid. The pseudo code of QeA-SOM is shown as follows:

Algorithm 1 QeA-SOM

1: $qelist \leftarrow \varnothing$
2: **for** $j = 1 \rightarrow k$ **do**
3: $\quad D_j \leftarrow D_{select} + D_{all}[j]$
4: $\quad M_j \leftarrow som\_make(D_j)$
5: $\quad qe_j \leftarrow quality(M_j, D_j)$
6: $\quad qelist \leftarrow qelist.add(qe_j, D_j)$
7: **end for**
8: $qelist \leftarrow assending\_sort(qelist)$
9: **return** $qelist$

The Algorithm 1 initializes a set *qelist* to be an empty set of Qe values at line 1. D_{select} and D_{all} are two inputs in the Algorithm 1. D_{select} is the set of metrics which have been selected as an input data for the aspect mining approach. D_{all} is the set of metrics which involving all the given metrics. The Algorithm 1 computes the Qe value from the first metric to the last one by using function *quality*. Function *som_make* [60] trains the input matrix data to the SOM data and function *quality* [60] computes Qe value for the j-th metric. The value of Qe will be added into the set of *qelist*. Finally, function *assending_sort* sorts the data set *qelist* according to the values of Qe in the ascending order.

3.3.2 Heuristic Algorithm QAHSSS

This section defines a QeA-SOM based heuristic metric selection algorithm, called QAHSSS (Algorithm 2), which is a variant of sequential forward selection (SFS).

QAHSSS algorithm takes a basic metric and a set of other given metrics which already removed the basic metric from them. The basic metric is a given metric which satisfies the formalized metrics. QAHSSS measures the quality for the selected set of metrics, which is defined as an input for aspect mining, in an iterative process. In each step, a metric is selected by the QeA-SOM algorithm and is moved from the given set of metrics into the selected set of metrics. The output of QAHSSS is a set of metrics which has the best quality. The quality in here is a quantization error (Qe) and topographic error (Te). A small value of Qe and Te is more desirable. The pseudo code of QAHSSS algorithm is shown as follows:

Algorithm 2 at the beginning initializes a set *sortlist* to be an empty set of Qe values, and initializes another set *qualitylist* to be an empty set of Qe and Te values. D_{init} represents the set of metrics which only involving basic metrics. D_init is trained by *som_make* sM into a SOM data sM and the quality is calculated for D_{init} at the beginning. After that the value of D_{init} is assigned to D_{select} and the set of given metrics is assigned D_{all}. The Algorithm 2 computes the evaluation criteria Qe and Te for the combination of metrics in an iterative process. In each step, a new metric which is selected by QeA_SOM

Algorithm 2 QAHSSS Algorithm

$sM \leftarrow \varnothing$
$sortlist \leftarrow \varnothing$
$qualitylist \leftarrow \varnothing$
$sM \leftarrow som\_make(D_{init})$
$[qe, te] \leftarrow som\_quality(sM)$
$qualitylist \leftarrow qualitylist.add(qe, te)$
$D_{all} \leftarrow D'$
$D_{select} \leftarrow D_{init}$
$index \leftarrow \varnothing$
repeat
    $sortlist \leftarrow QeA\_SOM(D_{select}, D_{all})$
    $j \leftarrow find(sortlist)$
    $D_{all} \leftarrow D_{all} - D_{all}[j]$
    $n \leftarrow n - 1$
    $D_{select} \leftarrow D_{select} + D_j$
    $M_{select} \leftarrow som\_make(D_{select})$
    $[qe, te] \leftarrow quality(M_{select})$
    $qualitylist \leftarrow qualitylist.add(qe, te)$
until $n = 0$
return $index = find(qualitylist)$

from D_{all} will be added into the D_{select} and that metric is removed from D_{all}. Function *som_make* [60] trains D_{select} to the SOM data M_{select}, and then function *quality* [60] calculate the value of Qe and Te for the SOM data M_{select}. The values of Qe and Te are added into the set of *qualitylist*. Finally QAHSSS algorithm returns the index of the step which measures the best quality for a set of metrics. The optimized metrics are the one which is used in the *index*-th step.

Chapter 4

Translating Name-based Pointcuts to Analysis-based Pointcuts

When refactoring the code bodies of crosscutting concerns into aspects, pointcuts are always generated using pointcut abstract techniques [10, 70]. As a name-based form, such as enumeration-pointcut that enumerates a set of explicit join points based on their specific names. Notice that name-based pointcuts rely on the names of fields, method, classes in the program, which are well known to be fragile against software evaluation. Analysis-based pointcuts on the other hand, rely on the properties of the program checkable via user-defined static program analysis, which therefore are free from the fragility. It makes, however, user hard to write programs because one has to write her analysis suitable for his properties.

To overcome the fragile pointcut problem and make analysis-based pointcut easy to use by programmers, we propose a framework in this chapter, namely Nataly. Nataly translates name-based pointcuts to analysis-based pointcuts automatically. We first introduce analysis-based pointcuts and illustrate how difficult to write analysis-based point manually with a concrete example in section 4.1. Next, we present the framework Nataly, which takes classes, name-based pointcuts as input and output is analysis-based pointcuts, in section 4.2. Nataly consists of four components, namely relationship analyzer (Section 4.2.1), seed generator (section 4.2.2), T-Pattern extractor (section 4.2.3 and section 4.2.4) and code generator (section 4.2.5).

4.1 Analysis-based Pointcut

Analysis-based pointcut [102] is one of the approaches to overcome the fragile pointcut problem. An analysis-based pointcut is defined by using static program analysis, and matches join points that satisfy the specific conditions which meet the intention of programmer. After the program has evolved, the source code details have to be changed. However, the specific conditions are not changed. In this case, the analysis-based pointcuts still work.

The question we have to ask here is although the analysis-based pointcuts avoid fragile pointcut problem, however, they are difficult to write manually. We assume that the programmer expects to use a regular expression in the analysis-based pointcut to match the join points. The programmer has to familiar with that regular expression. The following pointcut which matches any method call whose name consists of only lower case

character:

```
pointcut executeLowercase()
   :(execution(* *.*(..)))
   && if(thisJoinPoint.getSignature())
         .getName.matches("^[a-z]+$");
```

The first part of the pointcut matches any method execution. The second part uses a conditional (if) pointcut to implement the analysis. In AspectJ, the programmer can write any boolean expression in and if point in order to add arbitrary conditions to a pointcut.

This example uses Java and AspectJ reflection APIs in order to analyze the program. The special variable `thisJoinPoint` is an object that contains information about the join point, and the methods `getSignature()` and `getName()` retrieve signature and name of the method being executed.

Furthermore, Assuming that the programmer expects to use program structural or control flow reachability [102] as the user defined analysis, the implementation of analysis-based pointcuts should be more complex than the regular expression case. Therefore, the programmer writes analysis-based pointcuts manually is difficult and requires her has enough experience and knowledge.

4.2 Translation Framework-Nataly

In this section, we propose a framework Nataly, which translate the generated name-based pointcuts into analysis-based pointcuts in order to avoid fragile pointcut problem. In Nataly, name-based pointcuts are used as input and the properties of the join points which are matched by the input pointcuts, are expressed by a set of relationship graphs. Relationship describes the trace relationships between program elements such as method call, field declaration. Analysis-based pointcut matches join points with properties obtained by static program analysis. These properties are called as intention pattern (T-Pattern), which is the core concept of analysis-based pointcuts, are abstracted from relationship graphs in Nataly. The common/stable vertexes are kept in the intention pattern when abstract them from relationship graphs.

4.2.1 Relationships

The relationship analyzer analyzes six structural relationships between the program elements based on the static program analysis. We give the definition of six relationships as follows:

$$R_{declared} = \{(incls, cls), (\{retT, mn, args\}, cls)\}$$

$$R_{fset} = \{(\{delT, retT, mn, args\}, \{delT', fn\})\}$$

$$R_{fget} = \{(\{delT, retT, mn, args\}, \{delT', fn\})\}$$

$$R_{mcall} = \{(\{delT, retT, mn, args\}, \{retT', mn', args'\})\}$$

$$R_{mconcretize} = \{(\{delT, retT, mn, args\}, \{retT', mn', args'\})\}$$

$$R_{tconcretize} = \{(cls, pacls), (cls, ins)\}$$

Where $(incls, cls) \in R_{declared}$ means the class $incls$ (a.k.a an inner class) is declared by the class cls. $(\{retT,mn,args\},cls) \in R_{declared}$ means the method whose name is mn, argument types are $args$ and return type is $retT$, is declared by the class cls. $(\{delT,retT,mn,args\},\{delT',fn\}) \in R_{fset}$ means the method whose name is mn, argument type are $args$, return type is $retT$ and declared class name is $delT$, sets the field whose name is fn, declared class name is $delT'$. Similarly, $(\{delT,\ retT,\ mn,\ args\},\ \{delT',\ fn'\}) \in R_{fget}$ means a field whose value is gotten within a method. $(\{delT,retT,mn,args\},\{retT',mn',args'\}) \in R_{mcall}$ means the method whose name is mn, argument type are $args$, return type is $retT$ and declared class name is $delT$ that calls the method whose name is mn', argument types are $args'$ and return type is $retT'$. $(\{delT,retT,mn,args\},\{retT',mn',args'\}) \in R_{mconcretize}$ means the method whose name is mn, argument types are $args$, return type is $retT$ and declared class name is $delT$, overrides/implements the method whose name is mn', argument types are $args'$ and return type is $retT'$. $(cls,pacls) \in R_{tconcretize}$ means the class cls extends the class $pacls$ (a.n.a an super class). $(cls,ins) \in R_{tconcretize}$ means the class cls implements the interface ins.

For example, if the relationship is described as $R_{fset} = \{(\{Point, void\ setX(Int)\}, \{Point, x\})\}$, it represents method $Point.setX(Int)$ sets the value of the field $Point.x$. If the relationship is described as $R_{tconcretize} = \{(Point, FigureElement)\}$, it indicates that class $Point$ extends an abstract class $FigureElement$.

4.2.2 Relationship graphs

The set of relationship graphs corresponds to the join point shadows which are selected by a pointcut. The relationship graph describes six relationships among program elements. The root of each graph is a join point shadow which is selected by the traditional name-based pointcut. We give a definition of relationship graphs as follows and explain them with a concrete example.

Definition 2 (Relationship Graph (RG)) *Relationship Graph RG is a quadruple (V, E, ϕ, r), where V and E are finite sets and V is a set of vertices which represents a set of program elements. E is a set of edges, which represents structural relationships among program elements. $r \in V$ is a root vertex, and represents a join point shadow which is matched by a given traditional pointcut, and ϕ is a function with domain E and codomain $P_2(V) = V \times V$.*

In the pictorial representation of the relationship graph, $RG = (V, E, \phi, r)$ where

- $V = \{e_1, e_2, \ldots, e_n\}$, n is the number of vertices in RG.
- $r \in V$, r is the root vertex in RG.
- $E = \{r_1, r_2, \ldots, r_m\}$, m is the number of edges in RG.
- $\phi = \binom{E}{V}$

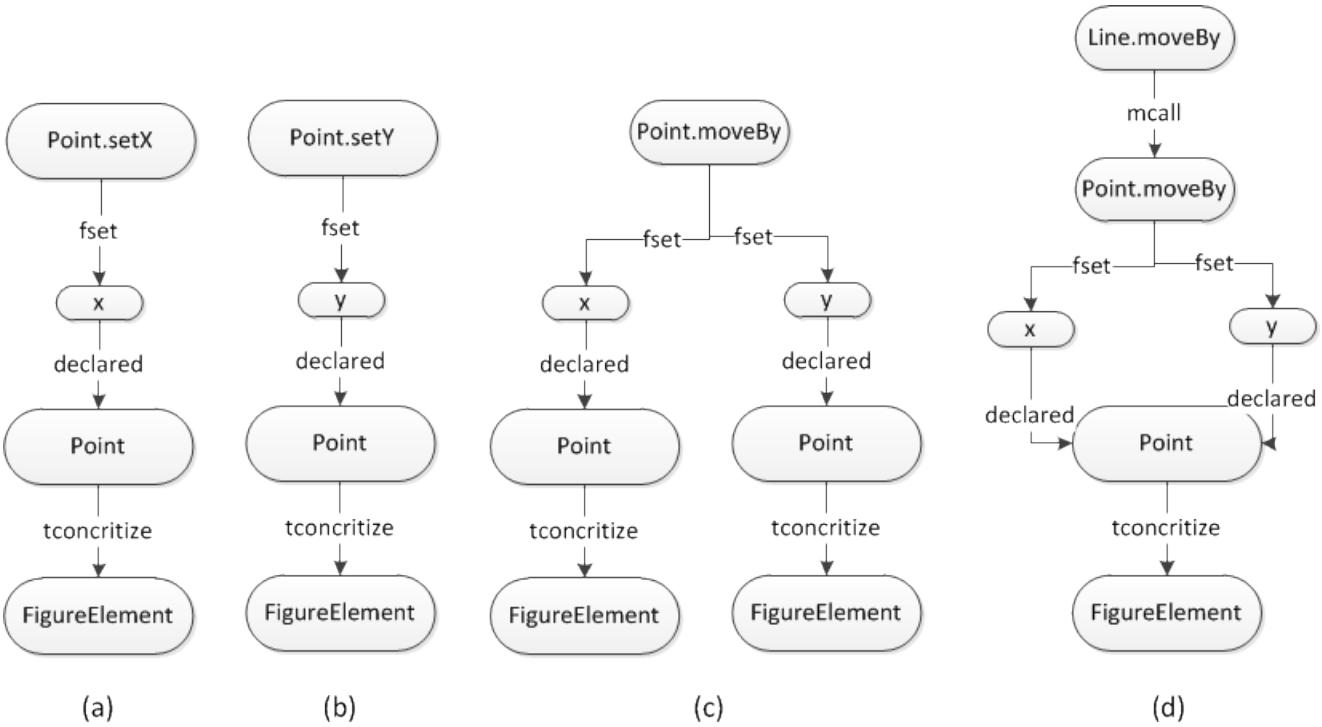

Figure 4.1: An example of Seed

Fig. 4.1 shows an example of a set of relationship graphs in the figure editor application. These graphs shown in (a) and (b) associate with two join point shadows: method execution of `Point.moveBy` and `Line.moveBy`, respectively, which are selected by the name-based pointcut "execution (* Point.move*(..)) || execution (* Line.move*(..))". These relationship graphs describe the following structural information: the method `Line.moveBy` calls the method `Point.moveBy` and the method `Point.moveBy` changes the value of field `x` and `y` simultaneously. Field `x` and field `y` are declared by the class `Point`. The class `Point` extends its super class `FigureElement`.

The pictorial definition of the relationship graph for the method `Line.moveBy` is described as follows:

$$RG = (V, E, \phi, r)$$

where

$$V = \{P.mv, L.mv, x, y, P, FE\}$$

$$r = L.mv$$

$$E = \{fset, mcall, declared, tconcretize\}$$

$$\phi = \begin{pmatrix} mcall & fset & fset & declared & declared & tconcretize \\ (L.mv, P.mv) & (P.mv, x) & (P.mv, y) & (x, P) & (y, P) & (P, FE) \end{pmatrix}$$

In this definition, Symbol $L.mv$ and $P.mv$ represent the method `Line.moveBy` and `Point.moveBy`, respectively; x represents the field `Point.x`; y represents the field `Point.y`; `P` represents the class `Point` and FE represents the `Point`'s super class `FigureElement`.

A significant point in the relationship graph is avoiding loops. The relationship between the program element and program element itself is not generated in our relationship

graph. The output (relationship graph) from the relationship analyzer is fed as input to the intention pattern extractor.

4.2.3 Intention Property

Intention property is a set of functions which derive from the paths of relationship graphs. Each function consists of an arc (relationship) and its head vertex (method and field).

We define intention property as $P_T(E_T) = V_T$. P_T is a function from arc (e.g., relationship) E_T and its head vertex (e.g., method and field) V_T. For example, the relationship between vertex e_m and e_n is r_n, then the function $P_n(r_n) = \{e_n\}$. We propose two types of intention properties: common property, and stable property.

Common property is a set of functions which represent the commonality within a set of relationship graphs. $P_{T_c} = P_1 \cap P_2 \cdots \cap P_m$ where m is the number of relationship graphs.

For example, assuming that the pointcut PC_A associates with two relationship graphs RG_1 and RG_2. Here, $RG_1 = (V_1, E_1, \phi_1, r_1)$, where

$$V_1 = \{M, A, B\}$$
$$E_1 = \{mcall\}$$
$$r_1 = M$$
$$\phi_1 = \begin{pmatrix} mcall & mcall \\ (M, A) & (A, B) \end{pmatrix}.$$

The relationship graph $RG_2 = (V_2, E_2, \phi_2, r_2)$, where

$$V_2 = \{N, C, B\}$$
$$E_2 = \{mcall\}$$
$$r_2 = N$$
$$\phi_2 = \begin{pmatrix} mcall & mcall \\ (N, C) & (C, B) \end{pmatrix}.$$

In this case, M is a method that calls method A, and method A calls method B. On the other side, method N calls method C, and method C calls method B. Both of the relationship graphs have a common function calling the method B. Therefore, $P_T(mcall) = \{B\}$ is the common property.

Stable property is a function P_{T_s} such that $P_{T_s}(mcall) = MN_s$ where its arc (relationship) is $mcall$ and its head vertex MN_s is a set of methods which are declared in the same abstract class or interface. We consider these methods have stable names. We say that a name is stable if it appears in an abstract class or an interface. Assuming that we add some new functions to a program, we rarely change the method in an interface. If we modify an abstract method, then all the methods in its sub-class must be changed.

Assuming, for example, that the pointcut PC_B associate with two relationship graphs RG_1 and RG_2. Here, $RG_1 = (V_1, E_1, \phi_1, r_1)$, where

$$V_1 = \{M.h, A.m1, F.m1, F\}$$
$$E_1 = \{mcall, mconcritize, declared\}$$

$$r_1 = M.h$$

$$\phi_1 = \begin{pmatrix} mcall & mconcritize & declared \\ (M.h, A.m1) & (A.m1, F.m1) & (F.m1, F) \end{pmatrix}.$$

The other relationship graph $RG_2 = (V_2, E_2, \phi_2, r_2)$, where

$$V_2 = \{N.g, B.m2, F.m2, F\}$$

$$E_2 = \{mcall, mconcritize, declared\}$$

$$r_2 = N.g$$

$$\phi_2 = \begin{pmatrix} mcall & mconcritize & declared \\ (N.g, B.m2) & (B.m2, F.m2) & (F.m2, F) \end{pmatrix}.$$

Here, method $M.h$ calls method $A.m1$, and method $A.m1$ implements method $F.m1$. Method $F.m1$ is declared in interface F. On the other side, method $N.g$ calls method $B.m2$, and method $B.m2$ implements method $F.m2$. Method $F.m2$ is declared in interface F. Method $A.m1$ and $B.m2$ have the stable names, hence the function $P_T(mcall) = \{A.m1, B.m2\}$ is the stable property.

We give a priority for the two types of intention properties. Assuming that the common property is not null, the intention property is equal to the common property; otherwise, if the stable property is not null, the intention property is equal to the stable property.

4.2.4 Intention Pattern

The Intention pattern (TP) is a flexible pattern which is used to match join points that meet the intention of the developer. Each intention pattern is abstracted from the paths of relationship graphs, which is associated with a traditional name-based pointcut. We define intention pattern as follows:

Definition 3 (Intention Pattern) *An intention pattern is a set of disjoint rooted trees: $TP = T_1 \cup T_2 \cdots \cup T_n$, n is the number of relationship graphs. T_i is a rooted Tree ($1 \leq i \leq n$) which is abstracted from the paths of relationship graphs. Each rooted Tree is a quadruple $T = (V', E', \phi, V_{?*})$, where $V_{?*} \in V'$ is a wild-card vertex, representing the root vertex which can be any program element. V' is a finite set of vertexes. $V' = \{V_{?*}, V_? \cup V_T\}$. Here, we import an element $V_?$, which is a wild-card vertex. $V_?$ represents the vertex which can be any program element. V_T is a set of vertexes within the intention property. E' is a finite set of arcs. $E' = \{r_?\} \cup E_T$, where $r_?$ is a wild card relationship, which represents any kind of relationship. E_T is a set of relationships within the intention property. ϕ is a function with domain E' and codomain $P_2(V') = V' \times V'$.*

In order to obtain a rooted tree, the intention property is kept and the program elements of class and interface are removed from the relationship graphs for each given set of relationship graphs. The rest certain program elements are replaced by wild-card $V_?$, and the root vertex is replaced by wild-card $V_{*?}$. Finally, the repeated paths are removed from relationship graph.

Algorithm 3 defines an algorithm that extracts intention pattern from the relationship graphs. We initialize an intention pattern Δ to be an empty set to be returned at line 1 and initialize a single pattern as an empty sequence of path. First, algorithm 3 builds

Algorithm 3 Extract Intention Pattern

Input: A set of relationship graphs for a given pointcut: R
Output: Intention Pattern Δ

1: $\Delta \leftarrow \varnothing$
2: $\delta \leftarrow <>$
3: {δ **is a single pattern**}
4: $B \leftarrow getCommonProperty(R)$
5: {B **is a set of common properties**}
6: **if** $B \neq \varnothing$ **then**
7: **for** $i \leftarrow 1$ to n **do**
8: {**n is the number of single common property** β **in** B}
9: $\delta_i \leftarrow (v)^{?*}$
10: $\beta_i \leftarrow B[i-1]$
11: $\delta_i \leftarrow \delta_i + ((r)^?, (v)^?)$
12: $\delta_i \leftarrow \delta_i + \beta_i$
13: $\Delta \leftarrow \Delta \cup \delta_i$
14: **end for**
15: **end if**
16: $A \leftarrow getStableProperty(R)$
17: {A **is a set of stable properties**}
18: **if** $A \neq \varnothing$ **then**
19: **for** $j \leftarrow 1$ to m **do**
20: {**m is the number of single common property** α **in** A}
21: $\delta_j \leftarrow (v)^{?*}$
22: $\alpha_j \leftarrow A[j-1]$
23: $\delta_j \leftarrow \delta_j + ((r)^?, (v)^?)$
24: $\delta_j \leftarrow \delta_j + \alpha_j$
25: **if** $\delta_j \notin \Delta$ **then**
26: $\Delta \leftarrow \Delta \cup \delta_j$
27: **end if**
28: **end for**
29: **end if**
30: **if** $A = \varnothing \wedge B = \varnothing$ **then**
31: **return** $\varnothing$
32: **else**
33: **return** Δ
34: **end if**

the pattern according to the common property. The algorithm `getCommonProperty(R)` returns a set of common properties. The symbol of $(v)^{?*}$ represents the root vertex. The symbol of $((r)^{?}, (v)^{?})$ represents a pair of vertex and its relationship. Such vertex and relationship represents any program elements and relationships, respectively. Second, algorithm 3 builds the intention pattern according to the stable property. The algorithm `getStableProperty(R)` returns a set of stable properties. Our algorithm will exclude the repeated single pattern from the intention pattern Δ. If our algorithm can not find any intention property then the algorithm will return an empty pattern.

Algorithm 4 Get Common Property

Input: A set of relationship graphs for a given pointcut: R
Output: A set of common properties B

1: $B \leftarrow \varnothing$
2: $r_r \leftarrow Random(R)$
3: $R' \leftarrow R - r_r$
4: $P \leftarrow PATH(r_r)$
5: **for all** $p \in P$ **do**
6: $\beta \leftarrow <>$
7: {β **is a single common property**}
8: **for** $i \leftarrow 1$ to n **do**
9: {**n is the number of** $< R, V >$ **pairs in path** p}
10: $\pi \leftarrow <>$
11: {π **is the** $< R, V >$ **property, initialize it as an empty pair**}
12: $\pi \leftarrow < (p)_i^r + (p)_i^v >$
13: **for all** $r \in R'$ **do**
14: **if** $\pi \notin r$ **then**
15: $\pi \leftarrow <>$
16: **break**
17: **end if**
18: **end for**
19: **if** $\pi \neq \varnothing$ **then**
20: $\beta \leftarrow \pi$
21: **end if**
22: **end for**
23: $B \leftarrow B \cup \beta$
24: **end for**
25: **return** B

Algorithm 4 defines an algorithm that find the common property from the relationship graphs. We initialize the set of common property as an empty set and initialize a single common property as an empty pair. $Random(R)$ randomly returns a relationship graph from the set of graphs. $PATH(r_r)$ divides the graph into a set of paths. $< (p)_i^r + (p)_i^v >$ represents the ith vertex in the path and its head relationship. Algorithm 4 traverses every $< R, V >$ pair in each path, and check whether the current pair exists in other relationship graph. If every relationship graph contains such pair, then it will be added into the set of common properties.

Algorithm 5 Get Stable Property

Input: A set of relationship graphs for a given pointcut: R
Output: A set of stable properties A

```
1: A ← ∅
2: for all r ∈ R do
3:    P ← PATH(r)
4:    for all p ∈ P do
5:       α ←<>
6:       {α is a single stable property}
7:       for i ← 1 to n do
8:          {n is the number of < R, V > pairs in path p}
9:          if (p)_i^r = R_mconcretize then
10:            if (p)_{i-1}^r = R_mcall then
11:               α ←< (p)_{i-1}^r + (p)_{i-1}^v >
12:            end if
13:         end if
14:      end for
15:      A ← A ∪ α
16:   end for
17: end for
18: return A
```

Algorithm 5 defines an algorithm that find the stable property from the relationship graphs. We initialize the set of stable property as an empty set and initialize a single stable property as an empty pair. Algorithm 5 traverses every relationship graph, and check whether the current graph contains a method which has a stable name. If it finds such method has a stable name, then such method and its head relationship will be added into the set of stable properties.

Fig. 4.2 shows an example of intention pattern which is abstracted from Fig. 4.1 . The symbol $V_{?*}$ in the ellipse indicates that the root vertex can be any program element. Symbol $V_?$ in the rectangle indicates that the vertex can be any program elements. Symbol $R_?$ on a line indicates that it can be any structural relationships. The example of intention pattern presents two situations: (a) indicates the relationship between the root vertex and vertex `x`, `y` should be `fset`; (b) indicates there is a vertex that can be any element as long as the relationship between it and vertex `x`, `y` is `fset`. In addition, the root vertex can reach this vertex regardless of the relationship.

We give a pictorial description of intention pattern TP_{ex} for this example as follows:

$$TP_{ex} = T_1 \cup T_2$$

$$T_1 = (V_1, E_1, \phi_1, V_{?*})$$

$$V_1 = \{V_{?*}, x, y\}, E_1 = \{fset\}$$

$$\phi_1 = \begin{pmatrix} fset & fset \\ (V_{?*}, x) & (V_{?*}, y) \end{pmatrix}$$

$$T_2 = (V_2, E_2, \phi_2, V_{?*})$$

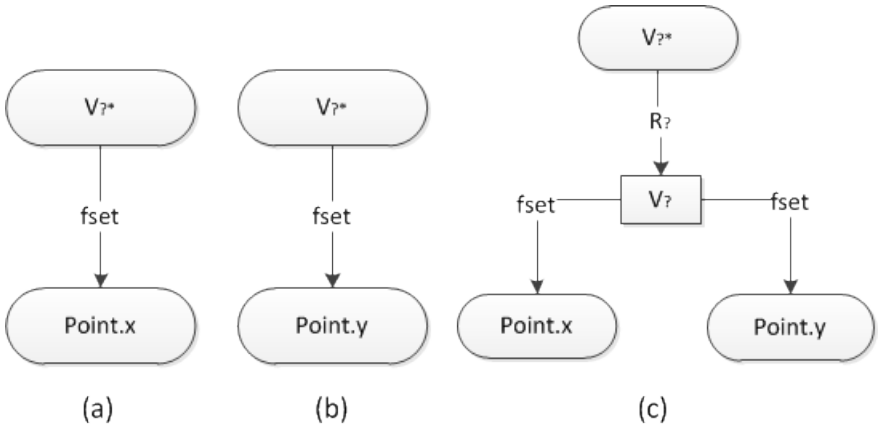

Figure 4.2: An example of T-Pattern

$$V_2 = \{V_{?*}, V_?, x, y\}, E_2 = \{R_?, fset\}$$

$$\phi_2 = \begin{pmatrix} R_? & fset & fset \\ (V_{?*}, V_?) & (V_?, x) & (V_?, y) \end{pmatrix}$$

When matching the join points with an intention pattern, we define a matching strategy: if the join point is matched by one of the rooted trees, then inferring that the join point is matched by the intention pattern, which includes that rooted tree.

Next, we illustrate how to match the join point with an intention pattern. Assuming that the intention pattern is TP=$(v_{?*} \xrightarrow{fset} x) \cup (v_{?*} \xrightarrow{fset} y)$. The given join point shadow is the method execution of `Point.setX`. Then the pictorial description of this relationship graph for this join point shadow is $RG_{ex} = (V_{ex}, E_{ex}, \phi_{ex}, setX)$:

$$V_{ex} = \{Point.setX, x\}, E_{ex} = \{fset\}$$

$$\phi_{ex} = \begin{pmatrix} fset \\ (Point.setX, x) \end{pmatrix}$$

In this example, the relationship graph for the join point shadow of method-execution `Point.setX` is $(Point.setX \xrightarrow{fset} x)$. The wild-card $v_{?*}$ can be replaced by the method $Point.setX$ in one of the rooted tree $((v_{?*} \xrightarrow{fset} x))$ within the intention pattern TP. In this case, the join point shadow of method-execution `Point.setX` is matched by the intention pattern.

4.2.5 Code generator

We propose an approach that can generate the code of analysis-based pointcut automatically. The code of analysis-based pointcut includes five parts:

1. Access specifier
2. Keyword (pointcut)
3. The name of analysis-based pointcut
4. Pointcut type

5. Pointcut signature

For the automatic generated analysis-based pointcut, the Access specifier, keyword and Pointcut type is not changed compare to the original name-based pointcut. The name of analysis-based pointcut is changed to a new one in order to distinguish the original one. The pointcut signature is changed to a wild-card which ignores the method or field name and their return type and arguments from the original one. Furthermore, a conditional pointcut designator is intersected into the code. The pointcut expression is appended with the conditional expressions such as "&&`(if(AMethod(jp)))`", where `AMethod` is an analysis method that is generated by our approach and `jp` is a join point shadow that needs to be matched. The conditional pointcut expression returns true if the join point shadow is matched with the intention pattern.

Following program slice shows a concrete example of an auto-generated analysis-based pointcut.

```
1 pointcut AnafigureChange()
2    :(execution(* FigureElement+.*(..)))
3    && if(isNeedfigureChange(JoinPoint));
4 boolean isNeedfigureChange(JoinPoint shadow){
5   TPattern pattern=new TPattern("figureChange");
6   TMatcher matcher=pattern.match(shadow);
7   return matcher.ismatch();
8 }
```

The pointcut `AnafigureChange` matches the method execution, which needs to repaint the canvas. A T-Pattern of `pattern` is initialized by using the key-word "figureChange". The T-Pattern matching module `matcher` checks the given join point whether it is matched by the T-Pattern. The method `isNeedfigureChange` returns true if the join point is matched by the T-Pattern. The class `TPattern`, `Tmatcher` are defined in the user library.

Chapter 5

Evaluation

In this Chapter, we shall discuss the usability, performance and effectiveness of our approach. We first give several case studies to show whether our approach can resolve the problem and then access the performance and usability of our approach via experiments.

5.1 Metrics Selection

This section discusses improvements on performance of clustering from the viewpoints of sensitivity, specificity and accuracy, which are introduced later. We evaluate the clustering-based aspect mining by using two Java applications. One is a banking system example in [32] and the other is a Java applet demo of Dijkstras shortest path algorithm [71]. Improvements on sensitivity, specificity and accuracy are evaluated by comparing the clustering data between optimized and original metrics. We compare the two clustering algorithms which are used in AMPA4C. We also analyze the program based on three different models (combination of metrics).

5.1.1 Benchmark

In order to provide a base for testing the methodology used and to determine whether our novel approach works, a number of suitable software requires to be selected as a benchmark. In this dissertation, we select two common software programs, which are frequently used in many research works. The Banking program [32] and Laffra's Djikstra's algorithm [71] are implemented as a benchmark test suit here. Table 5.1 lists some characteristic of these two programs. The Loc represents a count of the number of lines in the target elements that contain characters other than white space and comments. Efferent Couplings is represents the number of types inside the target elements that depend on types outside the target elements.

Banking Program

Banking program is frequently used in [32] as an aspect-oriented sample to explain how to use aspect techniques. The banking program has two crosscutting concerns: `logging before` and `logging after`. In a banking system, we would log each account transaction with information such as nature of the transaction, the account number, and the transaction amount. During the development cycle, logging plays a role like to a debugger. It is also usually the only reasonable choice for debugging distributed programs. By examining the log, a software engineer can spot unexpected system behavior and correct it. A log

Benchmark	Loc	Classes	Methods	Efferent Couplings	Crosscutting Concerns
Banking	328	14	35	10	logging before; logging after
Lafra's Dijkstra's Algorighm Application	890	7	42	6	show line; repaint; locking; unlocking

Table 5.1: Characteristics of Benchmark Programs

also helps the software engineer see the interaction between different parts of a software system in order to detect exactly where the problem might be.

Application of Dijkstra's Algorithm
The application of Laffra's Dijkstra algorithm (LDA) has four crosscutting concerns. The methods `showline` and `repaint` are discovered as crosscutting concerns in [58]. The method `showline` displays error messages when some preconditions are not met. The method `repaint` will be invoked when a new step is executed, or a new execution starts with new input data, or the algorithm is finished. Another two crosscutting concerns are `locking` and `unlocking`, which are discovered in [9]. They lock or unlock the user graphical interface each time when a new function was executed.

5.1.2 Metric Models

Model 1: Eleven metrics
Model 1 is a 11-dimensional vector space that consists of eleven metrics, which are frequently used in other aspect mining approach, The metrics are listed as follows: Method spread (MSP) [57], Fan-in value (FIV) [46], Fan-out value (FOV) [57], method internal coupling (MIC), method external coupling (MEC) [57], Affected classes (AC)[57, 58], Henry and Kafuras structure complexity (C_p) [72], return value(RV) [43], the number of parameters(NPM) [72], method signature(MSig) [57] and cohesion on method (COM) [72].

First, we give a brief explanation for each metric which is not mentioned before:

Method Spread (MSP): The metric measures level of interaction between the method and other classes. Modules with a small MSP value are expected to have less interaction between the method and classes in the software system. MSP is formally defined as follows:

$$MSP = \frac{\text{the number of classes which calls the given method}}{\text{total number of classes}}$$

Method Internal Coupling (MIC): Coupling is used to measure how dependent a software unit is to other units in a software system. The coupling would use to help differentiate between units of module. MIC is formally defined as follow:

$$MIC = \frac{|M_l|}{|M_l| + |M_e|}$$

where $|M_l|$ represents the number of all local calls to a method, $and|M_e|$ represents the number of all external calls to a method.

Method External Coupling (MEC): MEC measures how coupled a method is to other classes which do not declare this method. The metric shows whether the method is used frequently by external classes compare with its owner class. MEC is formally defined as follow:

$$MEC = \frac{|M_e|}{|M_l| + |M_e|}$$

where $|M_l|$ represents the number of all local calls to a method, $and|M_e|$ represents the number of all external calls to a method.

Return Value (RV) [43] represents the return value. If the value is 1, it represents the method has a return value. If the value is 0, it represents the method does not have a return value. **Number of parameter (NPM)** [74] represents the number of parameters within a given method. **Cohesion on method (COM)** [74] represents how many percent of the accessing local variables for a given method.

Model 2: Fan-in value and BVector

Model 2 consists of the fan-in value and BVectors $B_1, B_2, \cdots, B_m$ [14, 58], which represent whether each method is called by other methods in some classes. m is the number of classes in the given program. The value of B_i is defined as follows:

$$B_j = \begin{cases} 1 & \text{if } \exists M \in methods(C_j). \\ & \text{M calls } M_j \text{ in class } C_j \text{ that calls } M_i \\ 0 & \text{otherwise} \end{cases}$$

Model 3: SigTokens SigTokens [73] is the signature of tokens for each method. The name of the method will be split into a number of tokens (from 1 to k). Each token represents an attribute from A_1 to A_k. The vector has x-dimensional, where x represents the summation of all the unique tokens from all the methods after filtering out the duplicates and non-significant ones. For the method M_i the vector is defined as

$$M_i = \{O_{i_1}, O_{i_2}, \cdots, O_{i_m}\}$$

Where O_{i_n} is 1 if and only if the method m_i contains the attribute A_n. Otherwise O_{i_n} is 0 ($1 \leq n \leq m$).

5.1.3 Criteria of Evaluation

We evaluate the performance of aspect mining with respect to three criteria, namely sensitivity (sen), specificity (spe) and accuracy (acc).

We define the sets of $Correct_{cc}$, $Wrong_{cc}$, $Correct_{nc}$, $Wrong_{nc}$ as follows:

$$Correct_{cc} = Total_{cc} \cap Retrieval_{cc}$$

$$Wrong_{cc} = Total_{cc} \cap Retrieval_{nc}$$

$$Correct_{nc} = Total_{nc} \cap Retrieval_{nc}$$

$$Wrong_{nc} = Total_{nc} \cap Retrieval_{cc}$$

Where $Total_{cc}$ and $Total_{nc}$ are the sets includes all methods that are considered as crosscutting concerns and general concerns, respectively. $Retrieval_{cc}$ and $Retrieval_{nc}$ are obtained by using a clustering algorithm. They are considered as the set of crosscutting concerns and the set of general concerns, respectively.

We define four static definitions in order to calculate the sen, spe and acc.

- True positive (TP) generally represents the individual has the condition and tests positive for the condition. It is the number of $Correct_{cc}$ in this paper:

$$TP = |Correct_{cc}|$$

- True negative (TN) generally represents the individual does not have the condition and tests negative for the condition. It is the number of $Correct_{nc}$ in this paper:

$$TN = |Correct_{nc}|$$

- False positive (FP) generally represents the individual does not have the condition but tests positive for the condition. It is the number of $Wrong_{cc}$ in this paper:

$$FP = |Wrong_{cc}|$$

- False negative (FN) generally represents the individual has the condition but tests negative for the condition. It is the number of $Wrong_{nc}$ in this paper:

$$FN = |Wrong_{nc}|$$

The criteria include:

- Sensitivity:

$$sensitivity = \frac{TP}{TP + FN}$$

  Sensitivity relates to the test's ability to identify positive result. The sensitivity in this experiment is the proportion of concerns which are the crosscutting concerns which test positive for it.

- Specificity:

$$specificity = \frac{TN}{TN + FP}$$

  Specificity relates to the ability of the test to identify negative result. The specificity in this experiment is the proportion of crosscutting concerns which are non-crosscutting concerns which test negative for it.

- Accuracy:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

  Accuracy is the statistical measure of how well the crosscutting concerns and non-crosscutting concerns are correctly identified.

5.1.4 Results of Banking Program

We evaluate our approach based on banking program. Table 5.2 shows the metric selection result for Model 1 (We do not show the detail data of Model 2 and Model 3 here, due to a large number of steps). It shows the importance of selected metrics from `step 1` to `Step 10`. The `Step 2` gets the best result of Ge, which gets a relative smallest value.

Table 5.2: QUALITY OF METRICS FOR MODEL 1 IN BANKING

Step	Qe	Te	Ge
Step 1	0.06326	0.0741	0.00469
Step 2	0.0708	0.0370	0.00262
Step 3	0.1629	0	0
Step 4	0.2581	0.1111	0.02867
Step 5	0.2481	0	0
Step 6	0.4107	0	0
Step 7	0.6522	0.0370	0.02413
Step 8	0.8687	0	0
Step 9	1.1515	0	0
Step 10	1.2698	0.0370	0.04698

Table 5.3: EVALUATION OF BANKING EXAMPLE FOR HC

Metric Models	Original metrics			Optimized metrics		
	Sensitivity	Specificity	Accuracy	Sensitivity	Specificity	Accuracy
Model 1	100%	100%	100%	100%	100%	100%
Model 2	100%	4%	11.1%	100%	88%	88.89%
Model 3	100%	4%	11.1%	100%	100%	100%

$Ge = Te \times Qe$, it is a general error that reflects the trade-off between Qe and Te. The value of Te is zero represents the structure of map is the best one in this case.

Fig 5.1 shows the quality of the different combination of metrics for each step. If the point appears approximate in the bottom left region, it represents that the algorithm is better. Notice that in Figure 5.1, although step 3 and step 5 approximate to the left bottom region, however they are not an optimal solution, because the value of Ge in these two steps is 0 and the value of Qe are twice times larger than the value of Qe in `Step 2`. Hence, they are not the optimal solution. Instead, The `Step 2` (Te=0.0370 and Qe=0.0708) is the optimal solution with the metrics {FIV,MSP,AC}.

The evaluation results can be seen in Table 5.3 and Table 5.4. Table 5.3 shows the evaluation results which apply the hierarchical clustering algorithm (HC). Table 5.4 shows the evaluation results which apply the k-means clustering algorithm (KAM). Compare the data showed in these two tables, we find that the accuracy which are computed from the optimized metrics are better than the ones which are computed from the original metrics for both of HC and KAM. In addition, the KAM algorithm cannot obtain a stable cluster in the experiment when we use the original metrics, because the k-means algorithm has a problem when the data contains outliers. The irrelevant metrics which are contained in the original metrics will disturb the k-means algorithm to obtain a stable clustering. However, see from Table 5.4 we find that the KAM algorithm obtains a stable cluster by using the optimized metrics.

5.1.5 Results of Dijkstra's Algorithm Application

We evaluate our approach based on Dijkstra's algorithm application. Table 5.5 shows the importance of selected metrics from `Step 1` to `Step 10` for Model 1 (We do not show

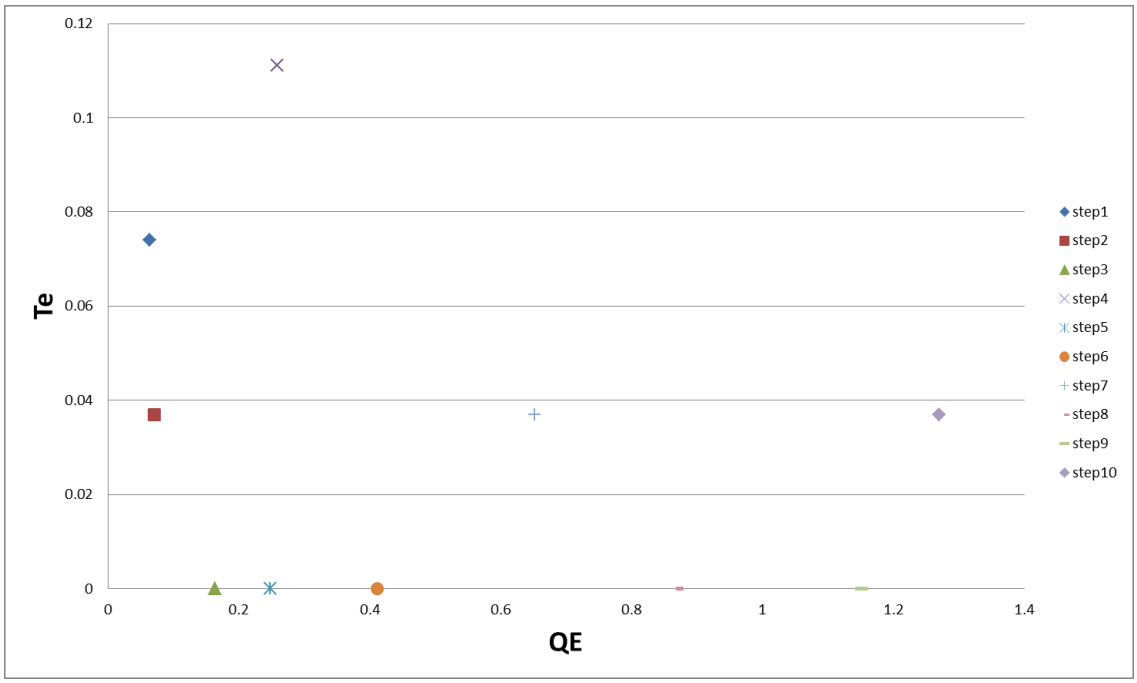

Figure 5.1: Quality of the different set of input metrics in banking program

Table 5.4: EVALUATION OF BANKING EXAMPLE FOR KAM

Metric Models	Original metrics			Optimized metrics		
	Sensitivity	Specificity	Accuracy	Sensitivity	Specificity	Accuracy
Model 1	NA			100%	92%	92.6%
Model 2	NA			100%	88%	88.89%
Model 3	NA			100%	100%	100%

the detail data of Model 2 and Model 3 here, due to a large number of steps). From `Step 1` to `Step 9`, all of the value of Te and Ge is 0, therefore the value of Qe becomes the most significant criteria of quality.

Fig 5.2 shows the quality of the different combination of metrics for each step in Laffra Djikstra's algorithm application. If the point appears approximate in the bottom left region, it represents that the algorithm is better. Notice that in Figure 5.2, both of `step 1` and `step 2` approximate to the left bottom region, because the number of metrics in `step 1` is smaller than the number of metrics in `step 2`, so that the `step 1` (Qe=0.0754 and Te=0.0) is the optimal solution with the metrics {FIV, MSP }.

For the LDA program the evaluation can be seen in the Table 5.6 and Table 5.7. Table 5.6 shows the evaluation which applies the hierarchical clustering algorithm. Table 5.7 shows the evaluation which applies the k-means clustering algorithm. Comparing Table 5.6 with Table 5.7, we find that the accuracy of clustering concerns which are computed from the optimized metrics are better than the ones which are computed from the original metrics for both of HC and KAM in the LDA program.

Table 5.5: QUALITY OF METRICS FOR MODEL1 IN LDA

Step	Qe	Te	Ge
Step 1	0.0754	0	0
Step 2	0.0754	0	0
Step 3	0.0968	0	0
Step 4	0.1985	0	0
Step 5	0.4300	0	0
Step 6	0.6020	0	0
Step 7	0.8294	0	0
Step 8	0.9094	0	0
Step 9	1.1869	0	0
Step 10	1.4859	0.0179	0.0266

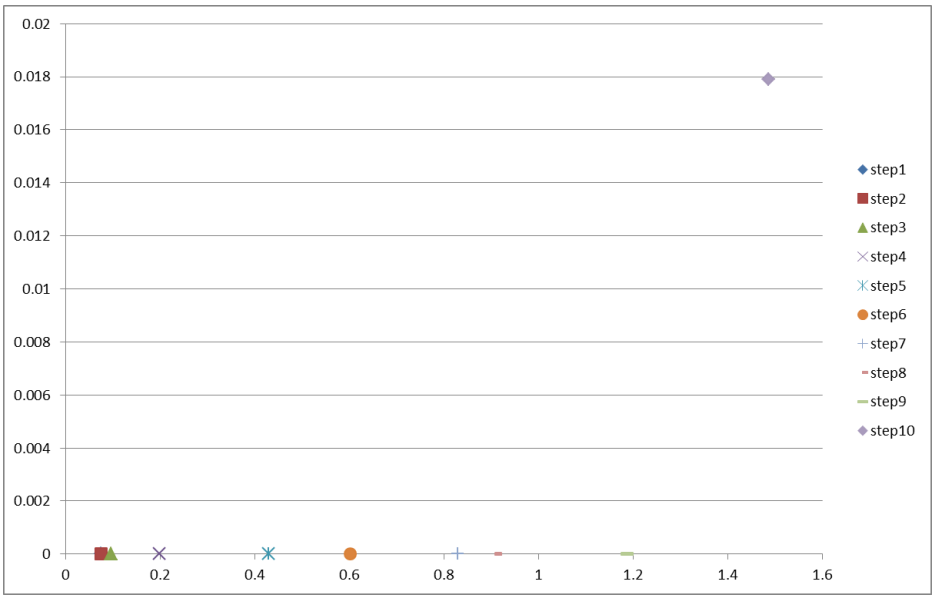

Figure 5.2: Quality of the different set of input metrics in LDA application

5.2 Framework Nataly

In this section, we first use a concrete example as a case study to explain how to extract Intention Pattern, and how to matching the join point shadows with Intention Pattern by using analysis-based pointcuts against software evolution.

Fig. 5.3 shows a case study of hybrid powered vehicle control system which has two power sources: a diesel engine and an electric engine. The method `increase(Fuel)` and `increase(Current)` increase the vehicle's speed with the `HybridAutomobile` notified to calculate the new speed.

We suppose that certain highways have a requirement that notifies vehicles of the speed limit. An aspect `SpeedingViolationPrevention` is used to control the vehicle's speed. The pointcut `speedChange` selects join points corresponding to the execution of `DieselEngine.increase(Fuel)` and `ElectricEngine.increase(Current)`. Both of the class `Fuel` and `Current` extend an abstract super class `Energy`. The wild-card `Energy+` represents the object references of type Energy and its subclasses. The traditional name-

Table 5.6: EVALUATION OF LDA FOR HC

Metric Models	Original metrics			Optimized metrics		
	Sensitivity	Specificity	Accuracy	Sensitivity	Specificity	Accuracy
Model 1	20%	100%	92.86%	20%	100%	92.86%
Model 2	20%	100%	92.86%	40%	100%	94.64%
Model 3	100%	2%	10.71%	20%	100%	92.86%

Table 5.7: EVALUATION OF LDA FOR KAM

Metric Models	Original metrics			Optimized metrics		
	Sensitivity	Specificity	Accuracy	Sensitivity	Specificity	Accuracy
Model 1	NA			100%	92.16%	92.86%
Model 2	NA			100%	68.63%	71.43%
Model 3	NA			100%	92.16%	92.86%

based pointcut is shown as follows. It matches the method-execution which its name text is `increase` and its parameter type is `Energy` or its subclasses.

```
aspect SpeedingViolationPrevention{
  pointcut limitspeed():
     execution(void increase(Energy+));
  //...
}
```

In this case study, we add a new requirement to the control system, which is about adding a new fuel cell energy source. In Fig. 5.4 a new class `FuelCell` is created. In order to increase the power, a numerical parameter is passed from `FuelCell` to a method `increase`. Intuitively, the aspect `SpeedingViolationPrevention` should also apply to the execution of this method. However, Parameter type double is a primitive type that does not satisfy the wild-card `increase(Energy+)`. This difference causes the pointcut does not match the method `FuelCell.increase(double)`.

Extracting Intention Pattern

The Intention Pattern derived from the relationship graphs for the traditional pointcut `limitspeed`, which is shown in Fig. 5.5. Both of methods `notifyChangeIn(Fuel)` and `nodifyChangeIn(Current)` set the value of field `overallSpeed`. Therefore, " the action of setting the value of `overallSpeed`" is a common property. The common property is $P_T(fset) = \{overallSpeed\}$, and the Intention Pattern for the analysis-based pointcut is $TP = (v_{?*} \xrightarrow{r_?} v_? \xrightarrow{fset} overallSpeed)$. It represents there is a vertex can be any program element as long as the relationship between it and field `overallSpeed` is `fset`. In addition, the rooted vertex can reach to this vertex regardless of the relationship.

Matching with Intention Pattern

The analysis-based pointcut `Analimitspeed` is shown as follows. The method `isNeedlimitSpeed` returns true if the join point is matched by the Intention Pattern.

```
  pointcut Analimitspeed():
     execution(* *.*(..))&&
     if(isNeedlimitSpeed(JoinPoint));
}
```

```
class HybridAutomobile{
  double overallSpeed;
  public void notifyChangeIn(Fuel fuel){
    this.overallSpeed+= fuel.calculateDeltaInMPH(this);
    //update attached observers...
  }
  public void notifyChangeIn(Current current){
    this.overallSpeed+= current.calculateDeltaInMPH(this);
    //update attached observers...
  }
  //...
}
class DieselEngine{
  HybridAutomobile car;
  public void increase(Fuel fuel){
    //...
    this.car.notifyChangeIn(fuel);
  }
}
class ElectricEngine{
  HybridAutomobile car;
  public void increase(Current fuel){
    //...
    this.car.notifyChangeIn(current);
  }
}
```

Figure 5.3: Hybrid automobile control system

Fig. 5.6 shows the relationship graph of the method-executions `increase(double)`. Assuming that we use the Intention Pattern ($v_{?*} \xrightarrow{r_?} v_? \xrightarrow{fset} overallSpeed$) to match this relationship graph. Here, the wild-card $V_{?*}$ can be replaced by the method increase(double), the relationship wild-card $r_?$ can be replaced by the relationship *mcall* and the vertex wild-card $V_?$ can be replaced by the method `nodifyChangeIn`. Therefore, the relationship graph of `increase(double)` is matched by the Intention Pattern. The analysis-based pointcut `Analimitspeed` can capture method `increase(double)`.

5.2.1 Experiment Result

In this section, in order to evaluate whether analysis-based pointcuts can improve the robustness of AspectJ application against software evolution, we compare the original name-based pointcuts and analysis-based pointcuts of open source applications over several releases in AspectJ.

```
class FuelCell{
  HybridAutomobile car;
  public void increase(double amount){
    //...
    Current current=this.generateCurrent(amount);
    this.car.notifyChangeIn(current);
  }
}
```

Figure 5.4: A new fuel cell class

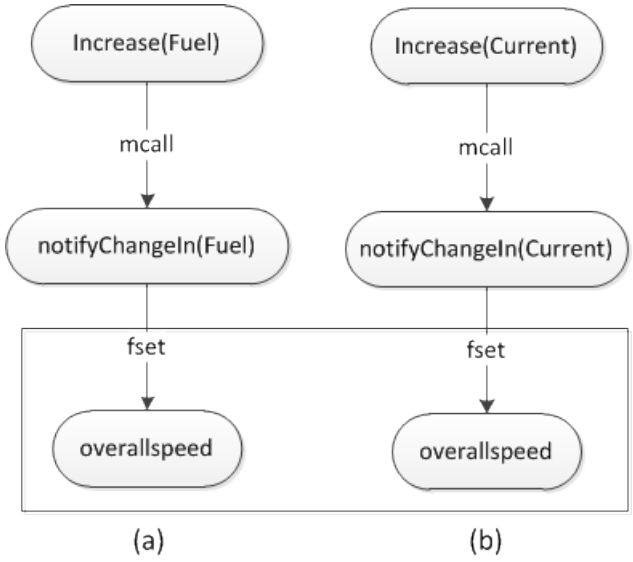

Figure 5.5: The seeds for pointcut limitspeed

Performance

All benchmark programs were executed on OpenSuse 12.1 with Intel Core2 CPU T700 2.0GHz and 2G memory. We executed each program for 5 times, and calculated the median values.

We evaluate the performance of NatalyAJ, which is an implementation of Nataly in Java, from micro benchmarks (e.g., Bean) to larger-scale benchmarks (e.g., Heath-Watcher). Table 5.8 shows the execution times of NatalyAJ for the 12 benchmarking programs. The columns titled **A**, **R** and **M** are the time elapsed for the three steps in our performance experiment. **A** represents the execution time of program analysis, **R** represents the execution time for abstracting the T-Patten and refactoring traditional pointcuts code to analysis-based pointcuts code. **M** represents the execution time for the join points matching with analysis-based pointcuts. Column **Total** represents the total execution time, which include analysis, refactoring, and matching, in seconds. The subjects along with an associated Line of code (**LOC**) range from 84 for Tracing to 9888 for JHotDraw. The number of classes (column **Class**) ranges from 4 to 222. The average execution time is 3.37 secs per KLOC and 180 ms per class. The result indicates that the time required to generate the analysis-based pointcuts and matching the join points are practical even for large applications.

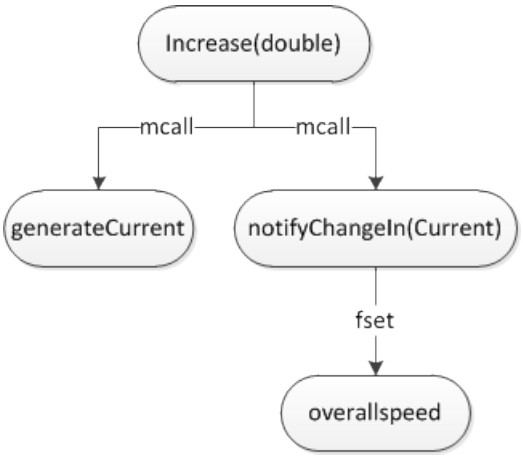

Figure 5.6: Relationship Graph for method increase(double)

Table 5.8: Execution times for NatalyAJ

subject	**LOC**	**Class**	**A** (ms)	**R** (ms)	**M** (ms)	**Total**(s)
MobileMedia	1209	28	2995	1875	2200	7
Telecom	290	14	1926	2019	548	4
Ants	951	34	2263	2147	5679	10
Bean	287	4	313	207	137	0.7
QuickSort	148	5	141	101	98	0.3
Spacewar	1434	37	2661	2418	974	6
Tetris	1076	26	980	841	321	2
Tollsystem	5194	150	5508	2017	5335	13
Tracing	84	4	109	142	87	0.3
JFTP	9724	65	10843	2585	12400	26
JHotDraw	9888	222	18129	3274	16775	38
HealthWatcher	5738	102	7795	1356	4991	14

Setup

We chose five open source applications written in Java, selected several versions from the archives, and translated the name-based pointcuts to analysis-based pointcuts using our approach. The software applications chosen were as follows:

1. Figure Editor is a simple application which supports to draw a figure on the canvas. We translate name-based pointcuts to analysis-based pointcuts which identify the code fragment that implements the Canvas repaint concern within the same aspect.

2. MobileMedia is a software product line for applications that manipulate photo, music and video on mobile devices. We translate the pointcuts within the aspects which implement the exception handling concerns.

3. JFtp is graphical FTP client software in Java for transferring files from one client to another. We translate the pointcuts within the aspects which implement GUI updating concern.

4. JHotDraw is a Java GUI framework for technical and structured graphics. We translate the pointcuts within the aspects which implement the Canvas repaint concern.

5. Health Watcher is a medium size web-based public health monitoring system. We translate the pointcuts within the aspects which implement the distribution concern.

For each version of the AspectJ application, we count the number of join point shadows in the next version that are accidentally captured and the number of join point shadows that are accidentally missed.

Results

Table 5.9: Number of matched join point shadows captured by pointcuts in different versions

		Accidental misses					Accidental capture			
	Version	**Total**	**MA**	**MD**	**MC**	**CHC**	**Total'**	**MA**	**MC**	**JPS**
Canvas repaint concern	v1	-	-	-	-	-	-	-	-	9
in Figure editor	v2	(2/8)	(2/6)	-	(0/1)	(0/1)	(0/2)	(0/1)	(0/1)	9
Exception handling concern	v1	-	-	-	-	-	-	-	-	24
in MobileMedia	v2	(0/3)	(0/1)	-	(0/2)	-	-	-	-	25
	v3	(4/4)	-	(4/4)	-	-	-	-	-	24
GUI updating concern	v1	-	-	-	-	-	-	-	-	16
in JFTP	v2	(0/6)	(0/6)	-	-	-	-	-		16
Canvas repaint concern	v1	-	-	-	-	-	-	-	-	4
in JHotDraw	v2	(1/1)	-	(1/1)	-	-	-	-	-	3
Distribution concern	v1	-	-	-	-	-	-	-	-	20
in Health Watcher	v2	(0/19)	-	-	-	(0/19)	-	-	-	20

Table 5.9 shows the number of accidental mismatches and accidental captures. There are five subjects, which are comprised of a series of discrete releases. Each row in the table corresponds to one version, indicating the following numbers.

- **Accidental misses:** The number (**a/n**) corresponds to the number of join point shadows which are missed by the analysis-based pointcuts and traditional name-based pointcuts, respectively. Following the total numbers of mismatches, the numbers in **MA**, **MC** and **CHC** columns show the breakdown by causes of the mismatches, which are explained below. Lower numbers are better.

- **Accidental capture:** The number (**a/n**) corresponds to the number of mismatched join points which are accidentally captured by the analysis-based pointcuts and traditional name-based pointcuts, respectively. Lower numbers are better.

- **JPS:** The number of intended join points which should be matched by each pointcut.

We classified the version of the accidentally missed and accidentally captured into the following categories and analyzed the effectiveness of analysis-based pointcuts for each category.

- **Method Addition (MA):** Adding a newly defined method that should be advised by an aspect in a new version of the application. In this scenario, because the traditional name-based pointcuts rely on the specific names of methods it cannot

capture the new method. On the contrary, analysis-based pointcuts can work as long as the new method is matched by the intention pattern. For example, a new method `updateImage` is added in version 2 of MobileMedia application, which updates the photo data in the database. The analysis-based pointcuts would capture this new method, because the new method `updateImage` is similar to the existing ones which execute the open and close operations on a store.

- **Method Deletion (MD):** Deleting a method which is captured by a pointcut in the previous version of the application. For example, method call of `Figure.invaliate()` is removed from method `addtoSelection(Figure)` in version 2 of JHotDraw application. In version 2, the join point shadow of method-call `Figure.invaliate()` cannot be captured by both of name-based pointcuts and analysis-based pointcuts, because the method no longer exists. Thus, neither of analysis-based pointcuts and name-based pointcuts can match this method in the new version of the application. We merely count the number of deleted methods in our experiment.

- **Method Change (MC):** Changing a method's signature in a new version of the application. For example, method `showImage()` and `showImageList(String)` in version 1 of MobileMedia application are changed to `showImage(String)` and `showImageList(String, boolean)` in version 2 of the application, respectively. The changed methods are mismatched by the name-based pointcuts because the text of method signature does not match the wild-card expression or the enumeration expression. On the contrary, the analysis-based pointcut captures these two methods `showImage(String)` and `showImageList(String, boolean)`, because the intention patterns are not changed.

- **Class Hierarchy Change (CHC):** Changing a class hierarchy in a new version of the application, such as inserting a newly defined class into a particular hierarchy or deleting a class from a particular hierarchy. For example, adding a new class `Circle` which extends the abstract class `FigureElement` in version 2 of Figure editor application. In version 2, method `moveBy(Int,Int)` in class `Circle` is mismatched by the name-based pointcut, because the signature of pointcut "`execution (* Circle.moveBy(..))`" is not enumerated in the traditional name-based pointcut. On the contrary, the analysis-based pointcut captures this method execution, because the new method `Circle.moveBy(Int,Int)` is similar to the existing one like `Line.moveBy(Int,Int)` which changes a figure element's visual property.

The results show that the fragile pointcut problem occurs frequently against software evolution. The five case studies exhibit the fragile pointcut problem when we use name-based pointcuts. The robustness of analysis-based pointcuts which are supported by our approach is higher than the traditional name-based pointcuts, because the analysis-based pointcuts match the join points based on intention properties instead of implementation details. For example, in the MobileMedia application the crosscutting concern of persistence-exception is applied when the method `showImage()`, `showImageList(String)` and `resetImageData()` execute. These methods are captured by a traditional pointcut `persistexception` with the enumeration expression. In the future version of this example, the developer will add a new method `updateImage()` in class `BaseController`. In

this case, the traditional name-based pointcut is broken, because the method signature of method `updateImage()` does not exist in the traditional pointcut's enumeration. Our approach automatic infers an intention pattern: $(V_{?*} \xrightarrow{R_?} V_? \xrightarrow{mcall} openRecordStroe) \cup (V_{?*} \xrightarrow{R_?} V_? \xrightarrow{mcall} closeRecordStore)$. This intention pattern indicates that the crosscutting concern of persistence-exception needs to be applied when the record-store is opened or closed. In the future version of this application, the analysis-based captures method `updateImage()` because method `updateImage()` calls the function of `openRecordStroe` and `closeRecordStore`.

Through the experiment, we found that the analysis-based pointcuts can improve robustness against software evolution, especially when the signature of a method changes, and when a new member/property (e.g., a method or a field) is added into a family.

On the other hand, we also found that generated analysis-based pointcuts may not be suitable to crosscutting concerns that are selectively applied based on some general non-functional properties, which are difficult to infer the intention patterns, such as general exception handling and efficiency.

In this section, we also demonstrate the usefulness of analysis-based pointcuts in a real-world setting. We evaluate the matching results for join points over releases in AspectJ with traditional name-based pointcuts and analysis-based pointcuts, respectively. In Table 5.9, there are four subjects, which are comprised of a series of discrete releases. The list **LOC** records the lines of code in the program. The list **class** record the number of classes inside the program. The list **jps** records the number of intended join points which should be matched by the pointcuts. The list **njps** records the number of join points which are actually matched by the traditional pointcuts. The list **ajps** records the number of join points which are actually matched by the analysis-based pointcuts. The list **f-jps** records the number of intended join points which is missed by the traditional pointcuts, but is captured by the analysis-based pointcuts.

Table 5.10: EXPERIMENT MATCHING RESULTS

subject	version	LOC	Class	jps	njps	ajps	f-jps
MobileMedia	1	1209	28	24	24	24	0
	2	1354	28	25	22	25	3
	3	1597	33	24	21	24	3
JFTP	1	9724	65	16	16	16	0
	2	11223	76	16	10	16	6
JHotDraw	1	9888	220	135	135	135	0
	2	28160	582	179	179	179	0
	3	28456	604	178	177	178	1
HealthWatcher	1	5738	102	87	87	87	0
	2	6122	112	87	56	87	21
	3	6769	132	87	56	87	21

Table 5.10 shows the empirical results produced by applying our approach on the evaluation subjects described previously. Columns 3-4 show the applications used for evaluation, the total lines of the Java source code and the number of class files in each application. Column **jps** shows the number of intended join points which should be matched by the pointcuts. Column **njps** and **ajps** shows the number of join point shadows which

are actually matched by the enumeration-based pointcuts and analysis-based pointcuts, respectively. Columns 8-9 show the number of join point shadows which should be captured, but is mismatched by the enumeration-based pointcuts and the analysis-based pointcuts, respectively.

In summary, the results in Table 5.9 and Table 5.10 show that there is a significant reduction in the number of mismatched join point shadows when using analysis-based pointcuts which are inferred by our approach. If using the straightforward enumeration-based pointcut, i.e., combining the specific pointcut expression of each join point shadow with a logical operator, then the number of mismatched join point shadows is high in the new version of application as indicated in Table 5.10. Through the experiment, we also found that the analysis-based pointcuts improve the robustness of aspect-oriented program against software evolution, especially when the signature of a method changes, and when a new member/property (e.g., a method or a field) is added into a family.

5.3 Concluding Remarks

For Aspect Mining:
The evaluation reveals two phenomena in the aspect mining experiment. Firstly, the KAM has better sensitivities than HC for all of the three metric models. For the LDA program, the evaluation which using KAM are better than the one which using HC when detect the crosscutting concerns. Hence, for the different program the optimal clustering algorithm is different. Secondly, by comparing the result of HC in the two case studies we find that the evaluation of HC is the best one for `Model 1` in banking program. However the evaluation of HC isn't the best one for `Model 1` in LDA program. The evaluation for `Model 2` is better than the evaluation for `Model 1` in LDA program. It reveals that an optimized metrics cannot suitable for all the programs. In other words, there is no existing reliable standard metric for aspect mining. Different programs have different optimized metrics and it is difficult to select them manually. Thus, it is essential to propose an approach that can remove the irrelevant metrics and optimize the original combination of metrics automatically.

For Aspect Interactions:
At the outset of this work, our study was to manually find the interference between aspects after we transform the implementation of crosscutting concerns from Java code to AspectJ code. First of all, we find that developer needs to understand the knowledge of AspectJ and be able to find all the advices which are advised at the same join point shadow. Second, the developer also needs to know the behavior of the whole program very well, since many details need to be handled. Otherwise, the developer is difficult to decide which aspect should have a higher precedence or which advice code cannot be inserted at a given join point shadow. Our current approach handled such issues, so that developers do not need to understand all the details of the program. Therefore, we believe that our approach will greatly improve the practicality of discovering and verifying aspect interference and will also remind developers to verify whether the behavior of their generating aspect-oriented program is probably changed or not.

For Framework Nataly:
We evaluate our approach on four different open source programs with three different versions. The comprehensible results show that the fragile pointcut problem occurs fre-

quently against program evolution. Depending on the evolution scenario, all of the four examples have the fragile problem when they use traditional name-based pointcuts. On the contrary, the analysis-based pointcuts work well in the future version of the program. The analysis data shown in the Table 5.9 represents that the analysis-based pointcuts with Intention Pattern which is generated by our framework are more robust than their counterpart traditional name-based pointcuts.

Chapter 6

Discussion for Aspect Interference

After detecting the crosscutting concerns from the legacy object-oriented program, the code bodies which implement the crosscutting concerns are removed from object-oriented code into aspects in the aspect refactoring phase. One of the issues that make aspect refactoring hard to be applied in practice is interference [66, 92, 91, 94, 95] among aspects. Such aspect interference is potentially dangerous and can result in erroneous behavior. If the behavior of aspect A reinforces the behavior of aspect B, the advice codes in aspect A should execute before the advice code in aspect B. For instance, Authorization and Authentication. Authentication verifies the identification of a user and Authorization establishes whether an authenticated user has sufficient privileges to access the system resources. Thus, authentication should execute before Authorization. When refactoring the function of authorization and authentication from object oriented program into aspects, the interference between such two aspects is still not dealt with. As a result, the compiler will free to choose the opposite order. If the execution order of such two aspects is changed, all of the users cannot access the system resources because all of them are unauthenticated when they access the system resources. In this case, such program is broken.

Obviously, checking the aspect interference is necessary in aspect oriented refactoring. However, it is difficult and inefficient to check such aspect interference manually for the developer. The reason is that (1) the developer needs to understand the knowledge of aspect oriented programming language very well. (2) The developer needs to analyze and understand the behavior of program very well, since many details need to be handled.

In the next section, we will show two scenarios to illustrate the problem of aspect interference and discuss the influence of such problem further.

6.1 Aspect interference

Aspect interference is derived from disordered aspect interactions. Aspect interactions, which arise in many software systems such as middleware and product lines , have been discussed by several groups of researchers [63, 64]. In the aspect-oriented program, aspects, which enable modularizing crosscutting concerns, are orthogonal to one another [65]. However, orthogonality between composed aspects may not be presumed in such case: aspects may reinforce one another for positive effect or, on the contrary, aspects may be incompatible for negative effect.

In order to understand aspect interference, Sanen et al. [67] have given a classification for aspect interactions, which classifies aspect interactions into four types: mutual exclusion, dependency, reinforcement and conflict, respectively. Dependency represents one aspect explicitly needs another aspect. Reinforcement represents one aspect offers extra support for another aspect and makes its extended functionalities become possible. Mutual exclusion and conflict represent the data/behavior which is added by one aspect, is incompatible with another and should be prevented. The difference is a conflict can be solved by mediation [68], however the mutual exclusion cannot. The interaction of dependency and reinforcement are positive aspect interactions. On the contrary, the interaction of mutual exclusion and conflict are negative aspect interactions.

If the interactions between the aspects are not dealt with when the developer refactor the code of crosscutting concerns into aspects, then the aspect interference potentially occurs. Moreover, it is also possible results in aspect interference due to the reorder of positive aspect interaction.

Next, we illustrate two different scenarios of aspect interference with two concrete examples. For each example, we also compare the different behavior of the program before and after the aspect refactoring. We assume that the potential codes of aspects have been detected. Additionally, such codes are moved from object-oriented codes into aspects which are written in AspectJ in each example. The aspect refactoring approach which is used in these examples is on the basis of Tools supported Refactoring [7].

6.1.1 Scenario 1

A Telecom application models a telecommunication administration system in Java. The base application includes two crosscutting concerns called Timing and Billing. Timing keeps track of the duration of a phone call, while Billing uses this information to calculate the amount money that the user are changed.

Fig. 6.1 shows class `Call`, which includes a method `hangup` that hangs up a call and drops connection. After the call is hanged up the `Timing` function and `Billing` function perform in sequence. The code statements in block `b1` implements the `Timing` function and the code statements in block `b2` implements the `Billing` function.

`Timing` function and `Billing` function (codes with underline) are refactored from class `Call` into aspects `Timing` and `Billing`, which are shown in Fig 6.2. Pointcut `endTiming` matches all the method-call join points for the methods which their names are the text of "drop" in class `Connection`. The instruction of after-advice in aspect `Timing` is executed after calling method `Connection.drop` in order to record the duration of a calling. The instruction of after-advice in aspect `Billing` is executed after the calling method `Connection.drop` in order to calculate the cost of a phone calling.

Depending on the precedence of two aspects, the code layout of the woven method will look like (a) and (b) in Fig 6.3. The Billing aspect works differently by the following reasons:

1. When the `Billing` aspect precedes the `Timing` aspect in (b). The billing advice will always receive 0 when read the duration time for a call because the duration time is recorded after calculating the cost. The behavior of the aspect-oriented program is totally different with the original object-oriented one.

```
Class Call{
 //...
 public void hangup(Customer c){
  for(Enumeration e=
    connections.elements();e.hasMoreElements();){
   Connection c=(Connection)e.nextElement();
   c.drop();
   //block b1 for the Timing function:
   getTimer(c).stop();
   c.getCaller().totalConnectTime+=
                  getTimer(c).getTime();
   c.getReceiver.totalConnectTime+=
                  getTimer(c).getTime();
   //
   //block b2 for the Bill function:
   long time=getTimer(c).getTime();
   long rate=conn.callRate();
   long cost=rate*time;
   getPayer(c).addCharge(cost);
   //
  }
 }
   //...
}
```

Figure 6.1: Outline of Class Call

```
1  Aspect Timing{
2   //...
3   pointcut endTiming(Connection c):
4     call(void Connection.drop()) &&
5     target(c);
6   after(Connection c):endTiming(c){
7     getTimer(c).stop();
8     c.getCaller().totalConnectTime+=
9                     getTimer(c).getTime();
10    c.getReceiver.totalConnectTime+=
11                    getTimer(c).getTime();
12  }
13 }
14 Aspect Billing{
15  //...
16  after(Connection c):Timing.endTiming(c){
17    long time=getTimer(c).getTime();
18    long rate=c.callRate();
19    long cost=rate*time;
20    getPayer(c).addCharge(cost);
21  }
22 }
```

Figure 6.2: Outline of Aspect Timing and Bill

Connection.drop
Timing(after proceed)
Billing(after proceed)

(a) When Timing precedes Billing

Connection.drop
Billing(after proceed)
Timing(after proceed)

(b) When Billing precedes Timing

Figure 6.3: Code layouts of the woven method in Telecom

2. When the `Timing` aspect precedes the `Billing` aspect. The duration time will be recorded first, and `Billing` can calculate the cost correctly. The behavior of the aspect-orient program is not changed.

6.1.2 Scenario 2

Supposing a search engine includes a number of search algorithms. Each algorithm includes two crosscutting concerns called performance monitor and logging. Performance monitor records the real-time execution time of the algorithm, while logging writes the input and output information of an algorithm into a log file.

Fig 6.4 shows class `BinaryAlgorithm`, which implements a binary search algorithm. Field `logging` is an instance of class Logger, which implements the logging function with method `log`. Method `run` calls the method `search` to find the object which its value is `v` from the given list `source`. Method `System.nanoTime` returns the current value of the most precise available system timer, in nanoseconds. Field `spendtime` represents the execution time of method `search`.

The code statements of performance monitoring and Logging are removed from class `BinaryAlgorithm` into aspects `LoggingAspect` and `MonitoringAspect`, which are shown in Fig 6.5 and Fig 6.6. Pointcut `p1` and `p2` match all the method-execution join points for the methods which their names are the text of "run" in class `BinaryAlgorithm`. The instruction of around-advice in `LoggingAspect` is executed around the method-execution `BinaryAlgorithm.run` in order to record the input and out message of method `search`. Similarly, the instruction of around-advice in `MonitoringAspect` is also executed around the method-execution `BinaryAlgorithm.run` in order to calculate the execution time of method `search`.

However, depending on the precedence of two aspects, the code layout of the woven method will look like (a) and (b) in Fig 6.7. The monitoring aspect works differently by the following reasons:

1. When the performance monitor precedes the logging aspect in (b). Performance monitoring traces an incorrect execution time of the search algorithm because the performance monitoring does not only trace the execution time of such search algorithm but also trace the execution time of logging. As a result, the outcome of performance monitoring is not correct.

2. When the logging aspect precedes the performance monitoring in (a). The logging aspect encloses the around advice performance monitoring. The monitoring only traces the execution time of the search algorithm. The behavior of the generating AO program is as same as the original object-oriented program.

```
Class BinaryAlgorithm{
  //...
  Logger logging;
  public void run(List source, String v){
    logging.log(source+v);
    long start=System.nanoTime();
    String result=search(source, v);
    long complete=System.nanoTime();
    long spendtime=complete-start;
    System.out.println("SpendTime: "+spendtime);
    logging.log(result);
    //...
  }
  //...
}
```

Figure 6.4: Outline of Class BinaryAlgorithm

```
Aspect LoggingAspect{
private Logger logging;
//...
Pointcut p1(String source, String v):
 execution(void BinaryAlgorithm.run(..))
 &&args(source,v);
 around(String source, String v):
  p1(source,v){
  logging.log(source+v);
  String result=
         proceed(source,v);
  logging.log(result);
  return result;
 }
}
```

Figure 6.5: Outline of Logging aspects

```
1  Aspect MonitoringAspect{
2  //...
3  pointcut p2(String source, String v):
4    execution(void BinaryAlgorithm.run(..)) &&
5    args(source) && args(v);
6  around(String source, String v): p2(source, v){
7    long start=System.nanoTime();
8    String result=proceed();
9    long complete=System.nanoTime();
10   long spendtime=complete-start;
11   System.out.println("SpendTime:"
12                      +spendtime);
13   return result;
14 }
15 //...
16 }
```

Figure 6.6: Outline of monitoring aspects

logging(before proceed)
monitoring(before proceed)
binarySearch
monitoring(after proceed)
logging(after proceed)

(a) When Logging precedes monitoring

monitoring(before proceed)
logging(before proceed)
binarySearch
logging(after proceed)
monitoring(after proceed)

(b) When Performance precedes logging

Figure 6.7: Code layouts of the woven method in search engine

All in all, when a developer moves the implementation of crosscutting concerns from an object-oriented program into aspects, the behavior of the generating aspect-oriented program may be changed and becomes unexpected. Therefore, it is essential to find a way to manage and detect the interference among such aspects in aspect-oriented refactoring.

6.2 Preliminary Idea

To check such aspect interference, an easy way is to walk through the execution order of the generating AO program and compare with the original OO program. However, there is a shortcoming in this way, which is we need to repeated walk through the entire program. It is difficult and inefficient. The reasons are (1) there are always too many repeated statements makes the execution order is difficult to analyze; (2) the code implementations of crosscutting concerns belong to several different aspects are inserted at the same location. It does not need to walk through the entire program; (3) if there are n number of aspects are inserted at the same location, the number of the permutation of such aspects will be $n!$. In this case, the developer requires to walk through the entire program for $n!$ times.

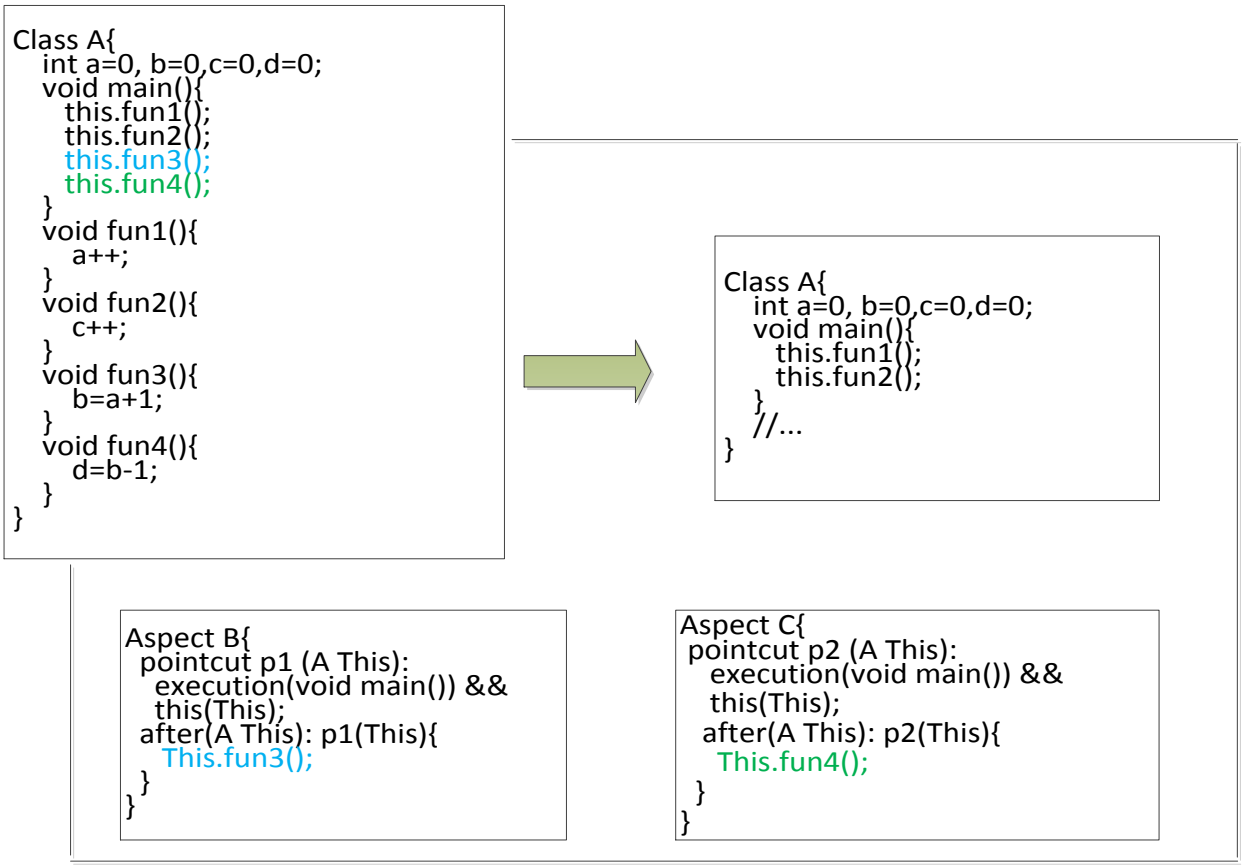

Figure 6.8: A simple example

In this section, we explain our idea for checking aspect interference efficiently in aspect-oriented refactoring on the basis of SMT. First, our approach extracts a model property `P` from the original OO program. Second, our approach extracts a model `M` for the generating AO program. If M $\models$ P (M models P), then there is no aspect interference occurs. If M $\not\models$ P (M does not model P), then it is possible to construct a counterexample showing where the property does not hold in the AO program. The aspect interference would be identified easily from such counterexample.

We use a simple example which is shown in Fig. 6.8 to explain our idea. We assume that in Fig. 6.8, there are two implementations of crosscutting concerns: `fun3` and `fun4`. Method `fun3` calculate the value of field `b` on the basis of field `a`. Similarly, method `fun4` calculates the value of field `d` on the basis of field `b`. In the aspect oriented refactoring, we moved the codes of crosscutting concerns from class A into aspects B and C, respectively.

6.2.1 Model Property for OO program

For the OO program, first, our approach generates a CFG on the basis of static control flow analysis for each advice holder. An advice holder is a method which a part of its code is removed from its body into aspects. For instance, class A in Fig. 6.8. Second, our approach translates such CFG into a model which is described by Conjunctive Normal Form (CNF). How to translate the control flow graph into a CNF model is shown in Fig. 6.9. Third, our approach extracts a property from such CNF model according to the control flow effective

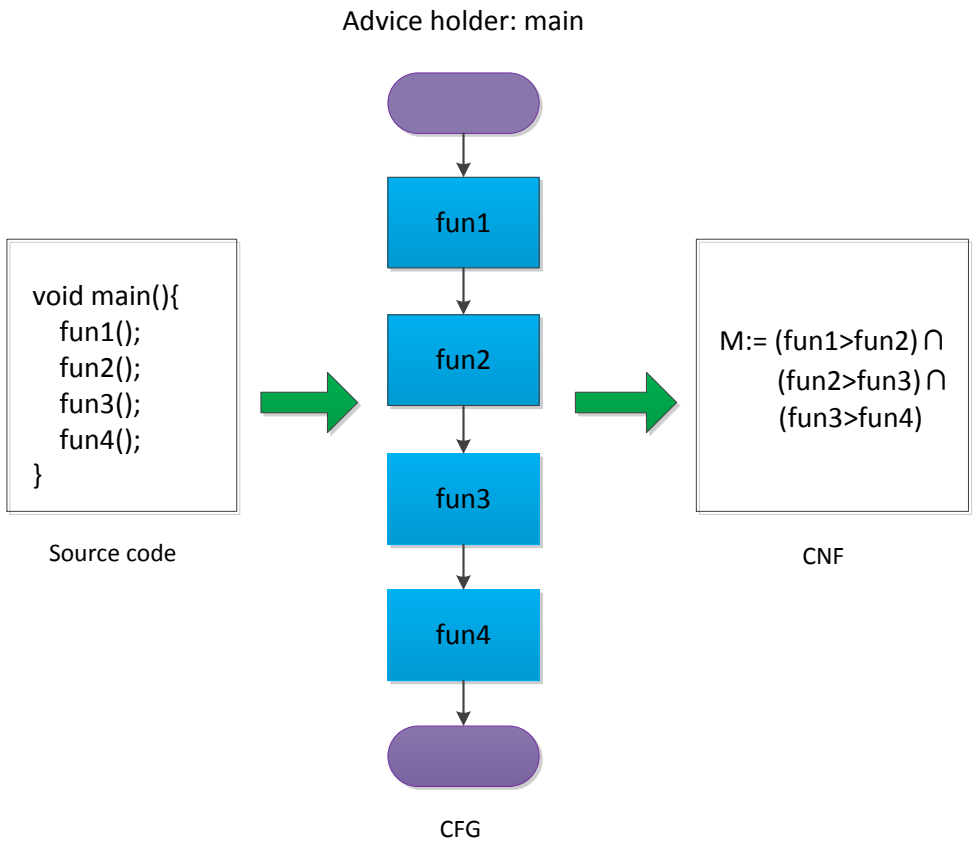

Figure 6.9: CFG and CNF Model for Simple Example (OOP)

statements. Control flow effective statements include (1) interdependent statements; (2) Exceptions which their `try` blocks contain the same code statements; (3)return statement; (4) Special system APIs, such as `System.exit`, `System.nanoTime`. The reason is because even such statements do not access the same resource, when the execution order of such statements is changed, they will still affect the behavior of the program.

How to judge two statements are independent is defined in the Definition 2. In the Definition 2, a statement is a tuple $s = (id, R_r, R_w)$. Where, id is the identifier of the statement s, R_r is the set of common resource (such as, variables, memory, files) which is read by the statement s, R_w is the set of common resource which is written or consumed by the statement s.

Definition 4 *Let two statements* $s = (id, R_r, R_w)$ *and* $s' = (id', R'_r, R'_w)$. *Independence of* s *and* s' *is:* $s \sqcap s' = \varnothing$, *where symbol* $\sqcap$ *represents get the accessed common resources of* s *and* s', *while symbol* $\varnothing$ *represents there is no accessed common resource.* $s \sqcap s' = (R_r \cap R'_w) \cup (R_w \cap R'_r) \cup (R_w \cap R'_w)$.

On the basis of the Definition 2, we construct a control effective table (IDT) for the object-oriented program. For example, the independence table for the simple example is shown in Fig. 6.10. Note that, in the IDT, "1" represents that two statements are control effective, "0" represents the two statements are not control effective.

Next, we design an algorithm to extract the model property on the basis of IDT from the CNF model. The algorithm is shown in Algorithm 6. In Algorithm 6, if two

	fun1	fun2	fun3	fun4
fun1	0	0	1	0
fun2	0	0	0	0
fun3	1	0	0	1
fun4	0	0	1	0

fun1: R_r={a} R_w={a}
fun2: R_r={c} R_w={c}
fun3: R_r={a} R_w={b}
fun4: R_r={b} R_w={d}

Figure 6.10: The IDT for Telecom example

Algorithm 6 : Model Property Extraction algorithm

Input: independent table IDT, a list of statements Π within CNF model τ (OOP)
Output: two sets of logic formulas S and S'

```
1: m:= the number of list Π
2: for (int j=1;j¡m;i++) do
3:     for (int k=j+1;k¡m;k++) do
4:         if IDT[j, k] = 1 then
5:             // statements Π_j and Π_k are not independent
6:             create logic formula: Π_j > Π_k
7:             S'.add(Π_j > Π_k)
8:         end if
9:     end for
10: end for
11: return
```

statements are not independent, then we create a logic formula for them. For example, we extract the model property P of simple example as follows:

$$P := (fun1 > fun3) \wedge (fun3 > fun4)$$

6.2.2 Model for AO program

For an object-oriented program, it is easy to obtain CFG with control flow analysis. However, for an aspect-oriented program, unlike explicit method call, the advice is inserted implicitly at a certain join point shadow. A join point shadow is a location in which advice-executing instructions can be inserted. Therefore, the traditional control flow analysis does not suit to analyze the aspect-oriented program (such as AspectJ application). Here, we select an interprocedural control flow analysis approach, which is first proposed in AJANA [20]. In our approach, we generate an interprocedural control flow graph (ICFG) for the generating aspect-oriented program that handles multiple advices at the same join point shadow.

Interprocedural control flow graphs (ICFGs) (1) include a standard CFG and also (2) interaction graphs (IGs) [69]. For multiple aspects, the IGs are built according to the precedence rules of the aspect-oriented program. If there is no precedence to reference, then the IGs are built for each permutation of advices which are inserted at the same join point shadow.

The example of ICFG is shown in Fig 6.11. There are two different types of ICFGs in Fig 6.11, because there is no precedence are specified for the aspect `B` and `C`. Thus the IGs are built for each permutation of advices.

Similarly, ICFG can be translated into CNF model. The example of how to translate the ICFG of the simple example to its CNF model is shown in Fig. 6.12 and Fig. 6.13. Note that the advice nodes are removed from the CNF model. Compare with the CNF model of the object-oriented program, there are two different CNF models are generated for the aspect-oriented program.

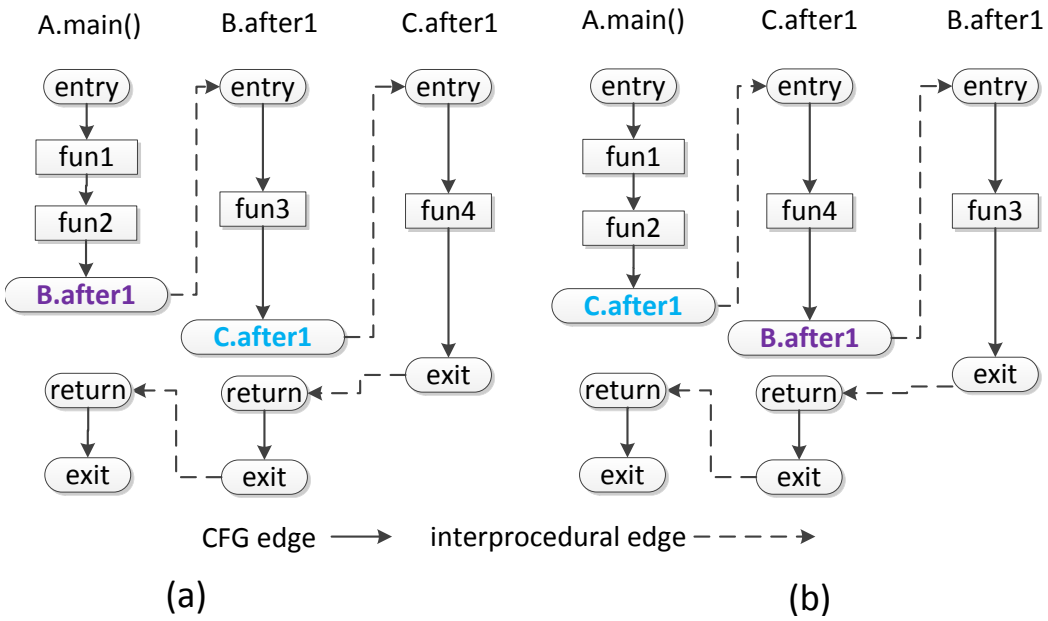

Figure 6.11: Interprocedural control flow graph for advice holder `hangup`

6.2.3 Checking Aspect Interference

In this section, the target of our approach is checking whether the generating aspect-oriented program has erroneous aspect interference or not. We assume that there is no erroneous interference among the implementation of crosscutting concerns in the original object-oriented program. Our approach applies a *Satisfiability Modulo Theories* (SMT) solver (such as Z3 [106], Yices [107]) to verify the CNF models. *Satisfiability Modulo Theories* (SMT) is an area of automated deduction that checks the satisfiability of logic formulas with respect to some logical theory [108]. If the property P does not hold in any of the models. As a result, erroneous aspect interference will be probably arising. In this case, our approach analyzes the counterexample and gives an awareness of such aspect interference for the developers.

For the simple example, our approach uses ($\neg P \wedge M_1$ (in Fig.6.12)) and ($\neg P \wedge M_2$ (in Fig.6.13)) as inputs, respectively, for the SMT solver. The SMT solver returns that the formula of ($M_2 \wedge \neg P$) is satisfiable. It indicates that the property P does not hold in M_2. In this case, SMT solver also returns a counterexample (for instance, $\{(fun4 > fun3)\}$). When we analyze this case, we find that the code statements of `fun3` are inserted after the code statements of `fun4`. As a result, when the program executes, the value of field d will be always receive -1, because the value of field d should be always 0. The behavior of such generating aspect-oriented program is totally changed.

6.3 Further discussion on case studies

In this section, we discuss on three case studies. To show how our idea works, we also give a concrete interprocedural control flow graph and conjunctive normal form in the first case study.

Telecom Application. As a typical example for an aspect interaction problem, inspired by an example of Telecom application, which has been introduced in Chapter 4 as

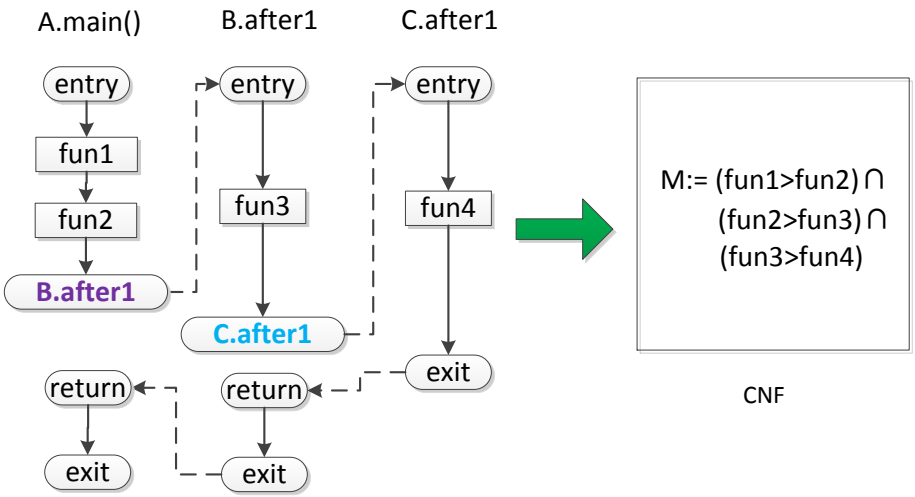

Figure 6.12: CNF for the AO program of simple example (a)

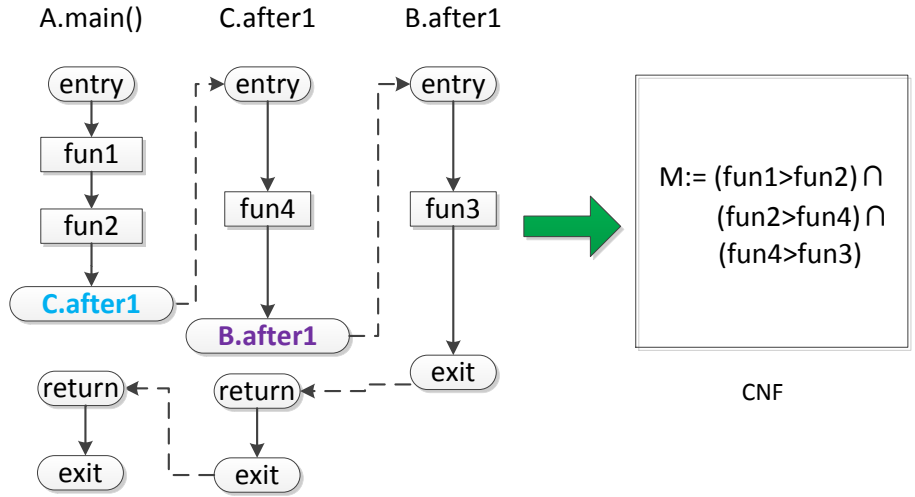

Figure 6.13: CNF for the AO program of simple example (b)

the first example to explain the aspect interaction problem. Here, we use this application as a case study to evaluate our approach.

In the aspect refactoring process the crosscutting concerns' implementation of Timing and Billing are removed from the Java code into aspects Timing and Billing, respectively. A partial of code is shown in Fig. 6.14.

Method `hangup` hangs up a user's call and drops connection. After method `drop` is called, the implementation of crosscutting concern *Timing* and *Billing* are executed in sequence. The call to method `getTimer(conn).stop` (id3) and the following two assignment statements (id5 and id6) are removed from the body of `hangup`. A new aspect, named `Timing`, is created to intercept the execution of `hangup` and insert id3, id5 and id6 after call method `Connection.drop()`. Similarly, the statements id7, id8, id9 and id10 are also removed from the body of `hangup` . A new aspect, named `Billing`, is created to intercept the execution of `hangup` and insert such statements after call method `Connection.drop()`.

Note that the precedence of the two aspects (Timing and Billing) is not specified and both of the statements in aspect Timing and Billing are inserted after call method

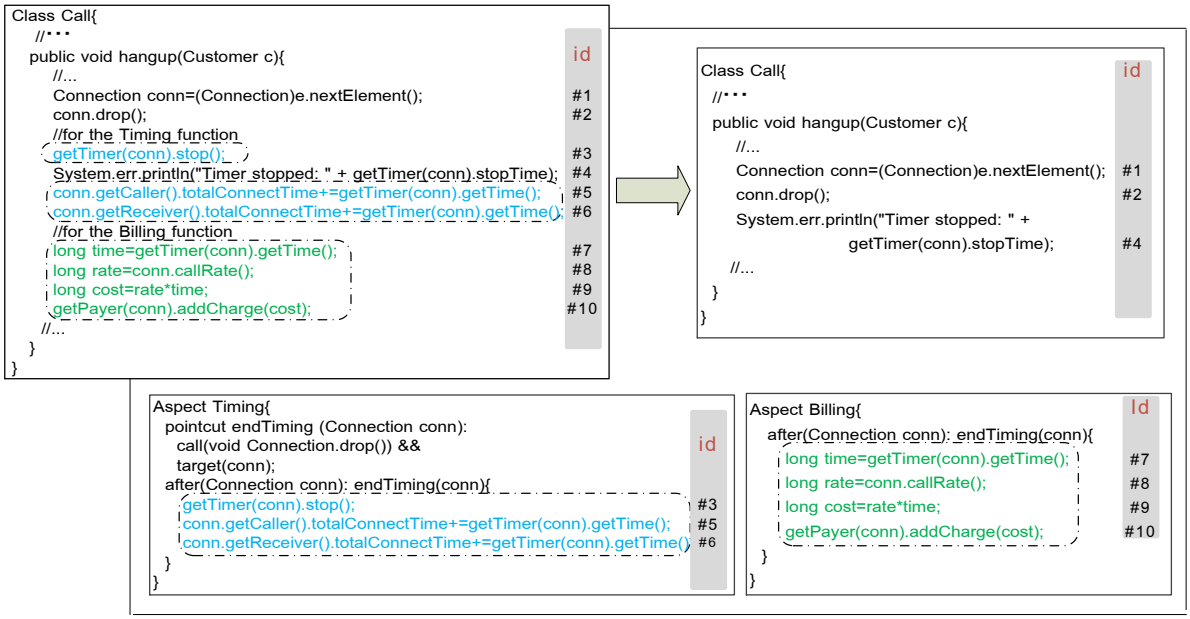

Figure 6.14: Example of refactoring: Timing and Billing

`Connection.drop()`. Thus, there should be two possible cases of the AspectJ program.

We analyze the Java code of Telecom application by static control flow analysis, and obtain the CNF model from the control flow graph. The CNF model is shown in Fig. 6.15. The algorithm 1 extracts the model property P from such control flow graph as follows:

$$\begin{aligned} P := & (\#1 > \#2) \wedge (\#1 > \#3) \wedge \\ & (\#1 > \#4) \wedge (\#3 > \#5) \wedge \\ & (\#5 > \#6) \wedge (\#3 > \#7) \wedge \\ & (\#3 > \#8) \wedge (\#7 > \#9) \wedge \\ & (\#8 > \#9) \wedge (\#9 > \#10) \end{aligned}$$

Next, we analyze the AspectJ code of Telecom application, and obtains the ICFG which is shown in Fig. 6.16. The CNF models which are extract from such ICFGs are shown in Fig. 6.17 and Fig. 6.18.

In the case (a), we use $(\neg P \wedge M_a)$ as inputs for the SMT solver. The SMT solver returns that the formula of $(\neg P \wedge M_a)$ is unsatisfiable. It indicates that the property P holds in model M_a. There is no aspect interference in the generating AO program of Telecom application.

In the case (b), we use $(\neg P \wedge M_b)$ as inputs for the SMT solver. The SMT solver returns that the formula of $(\neg P \wedge M_b)$ is satisfiable. It indicates that the property P does not hold in model M_b. SMT solver also returns a counterexample (for instance $\#7 > \#6$).

We analyze the case (b), we find that the code statements of Timing (such as #6) are inserted after the code statements of Billing (such as #7). As a result, when the program executes, the code of Billing will be always receive 0 when read the value of duration time for a user call. The behavior of such generating aspect-oriented program

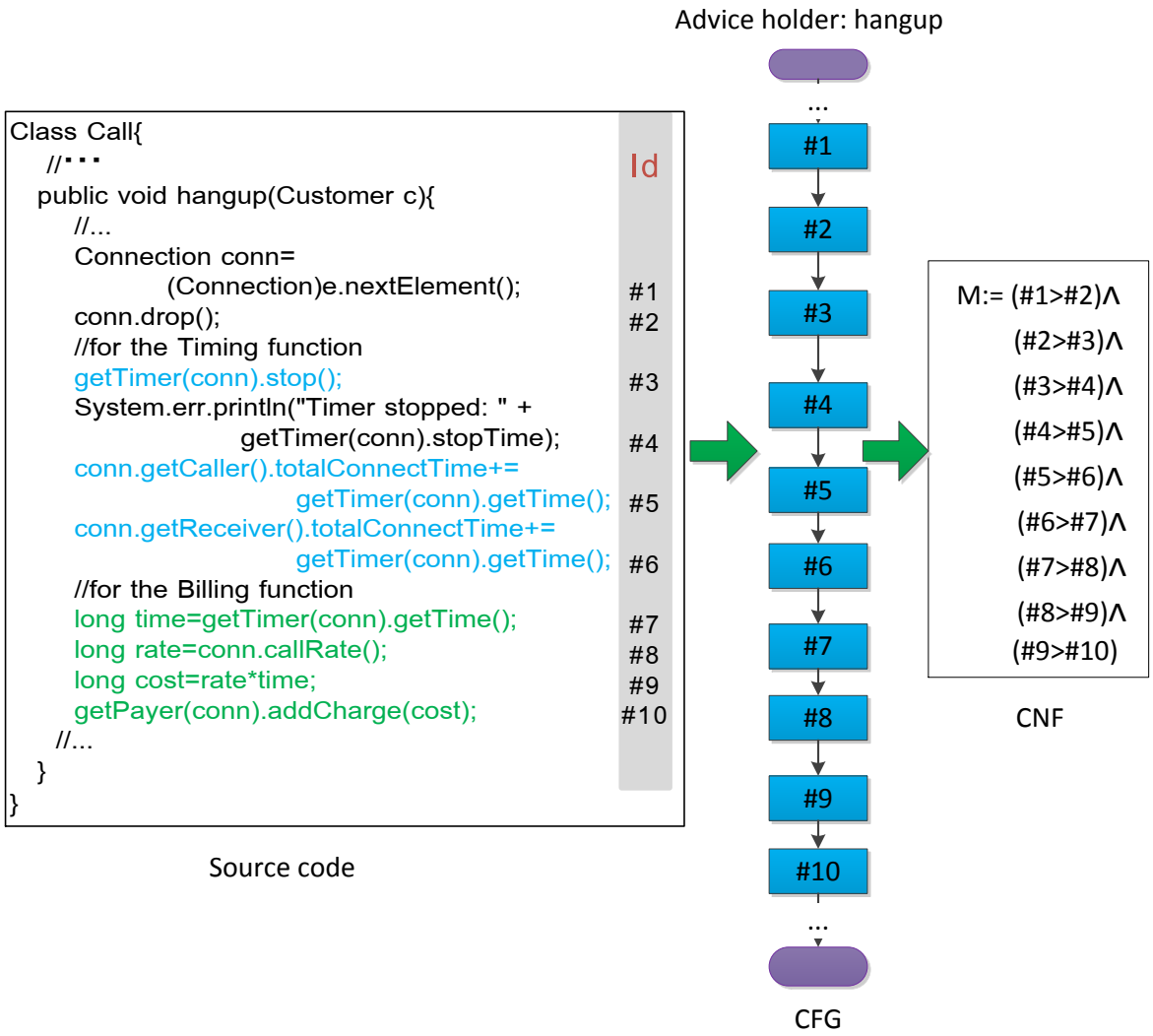

Figure 6.15: CFG and CNF model for Telecom (OOP)

is totally changed. To ensure the behavior of such program is not changed, the aspect Billing should have a higher precedence than aspect Timing in AspectJ version.

From this case study we observe that our approach verifies the aspect interference successfully and also finds a suggested solution for the given Telecom application.

Health watcher The main goal of this second case study was to assess whether our approach could verify the interference between the aspects of exception handler. The health watcher is a web based system designed to monitor public-health-related complaints and notifications. By allowing people to register several kinds of health complaints, health care institutions can promptly investigate the complaints and take the required action. There are several crosscutting concerns in the Java implementation of this system, such as command pattern, exception handling distribution and concurrency. The code implementations of these crosscutting concerns are removed from Java code to the aspect code.

In the aspect refactoring process the crosscutting concerns' implementation of exception handle and command pattern are removed from the Java code into aspects `HWTransactionExceptionHandler` and `ServletCommanding`, respectively. A partial of code is shown in Fig. 6.19.

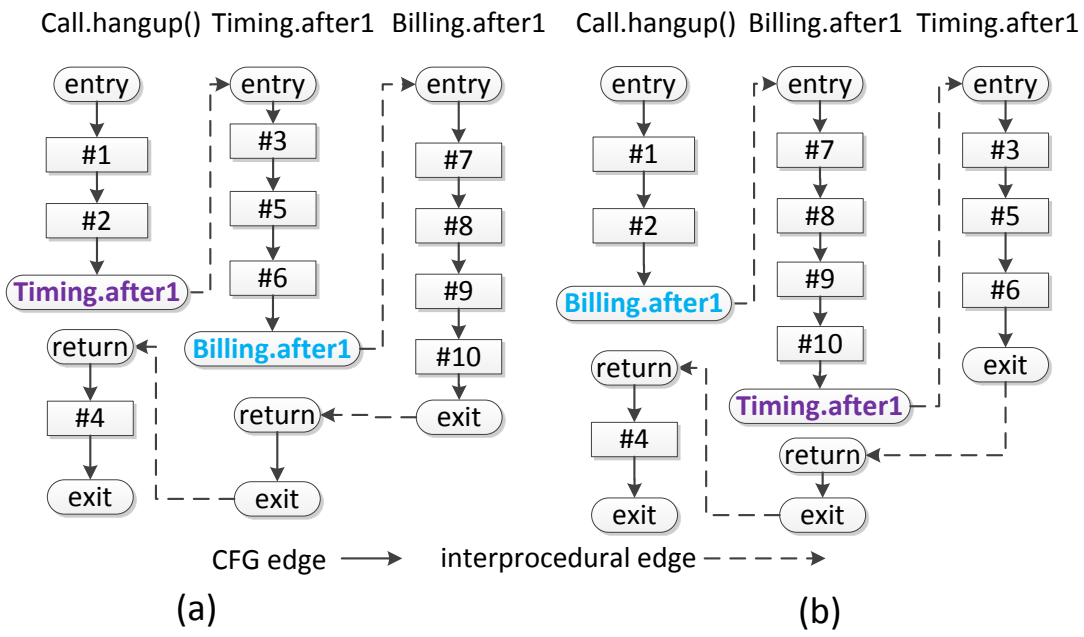

Figure 6.16: Interprocedural control flow graph for Telecom

In the original Java implementation, the exception handling handles the operation of command pattern. When we refactor such code implementation of crosscutting concerns into aspects, the interactions between them is not dealt with. The precedence of such aspects does not be specified. Thus the weaver is free to choose the opposite order, as in this case advice precedence according to the language specification in undefined. As a consequence, the exception handling does not happen around the command pattern. If an exception happens in the command pattern, then this exception cannot be caught and handled. Program behavior is changed and program function is broken.

To address the interactions between the two aspects, we use our approach to verify the aspect interference. There are three before advices, one after advices and five around advices in the AspectJ program. For the aspect `ServletCommanding` (command pattern) and aspect `HWTransactionExceptionHandler` (exception handling), there are two locations that such aspects' advice codes are inserted at the same time. The SMT solver returns that the one of the formula of $(\neg P \wedge M)$ is satisfiable. It indicates that the property P does not hold in model M. We find that in this case, the code statements of transaction exception handler do not around the code statements of command pattern. As a result, the exception cannot be caught and handled in the AspectJ version. This case study allows us to verify an aspect interference of reinforcement between two aspects by our approach.

Search Engine. Search Engine is a module which supports a number of search algorithms. Each algorithm includes two crosscutting concerns called performance monitor and logging. Performance monitor records the real-time execution time of the algorithm, while logging record some information of that algorithm into a log file.

Fig 6.20 shows class `BinaryAlgorithm`, which implements a binary search algorithm. Field `logging` is an instance of class Logger, which implements the logging function with method `log`. Method `run` calls the method `bin_sch` to find the object which its value is `v` from a String list `source`. Method `System.nanoTime` returns the current value of the most precise available system timer, in nanoseconds. Field `spendtime` represents the

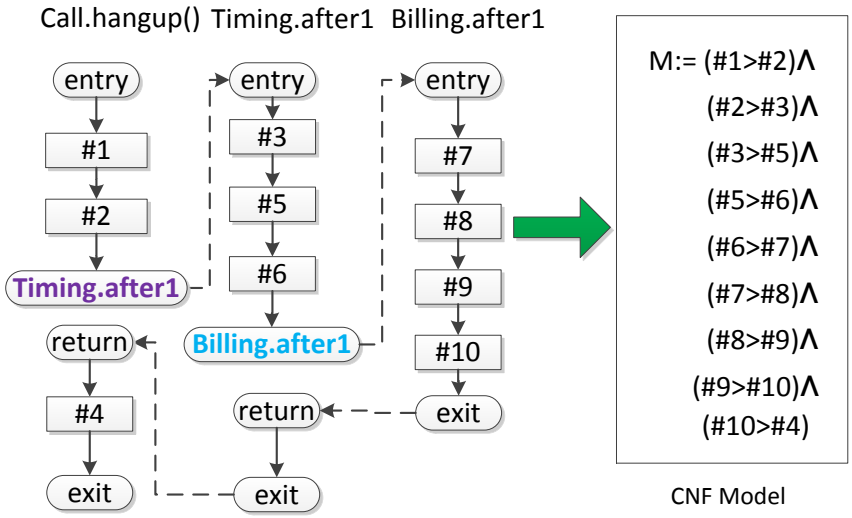

Figure 6.17: CNF for the AO program of Telecom (a)

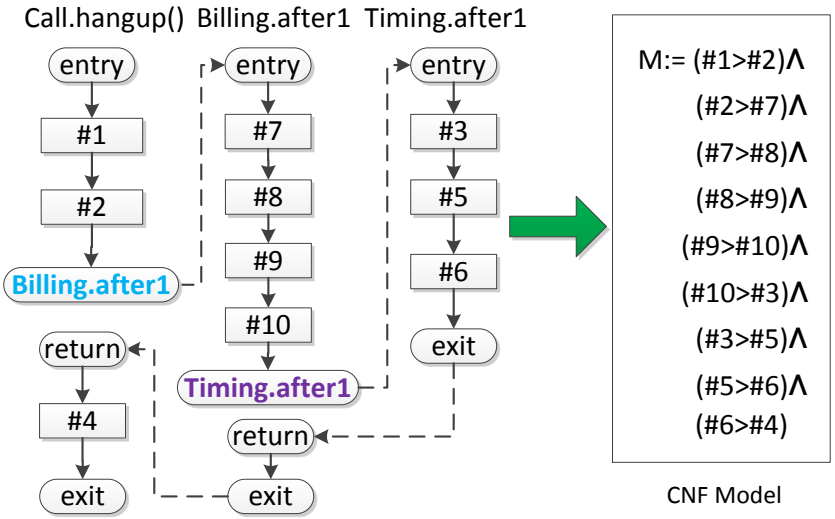

Figure 6.18: CNF for the AO program of Telecom (b)

execution time of method `bin_sch`.

The code implementation of performance monitor and logging are removed from Java code into aspects LogAspect and Monitor, respectively. Note that the statement `id1` and `id2` will be inserted at the same location, while `id4`, `id5`, `id6` and `id7` will be inserted at the same location when the weaver weaves the aspect-oriented program. However, the precedence is not specified. Thus, the weaver is free to choose the insert order. When the aspect performance monitor precedes the aspect logging performance monitor traces an incorrect execution time of the search algorithm because the performance monitor not only traces the execution time of the search algorithm but also traces the execution time of logging. As a result, the results of performance monitor are not correct.

In this case, we define the system function `System.nanoTime` are not independence with other code statements which execute within the method `run`. Therefore, the SMT solver returns that one of the formula of $(\neg P \wedge M)$ is satisfiable. It indicates that the property P does not hold in model M. We find that in this case, the code statements of logging do not around the code statements of monitoring. As a result, the performance

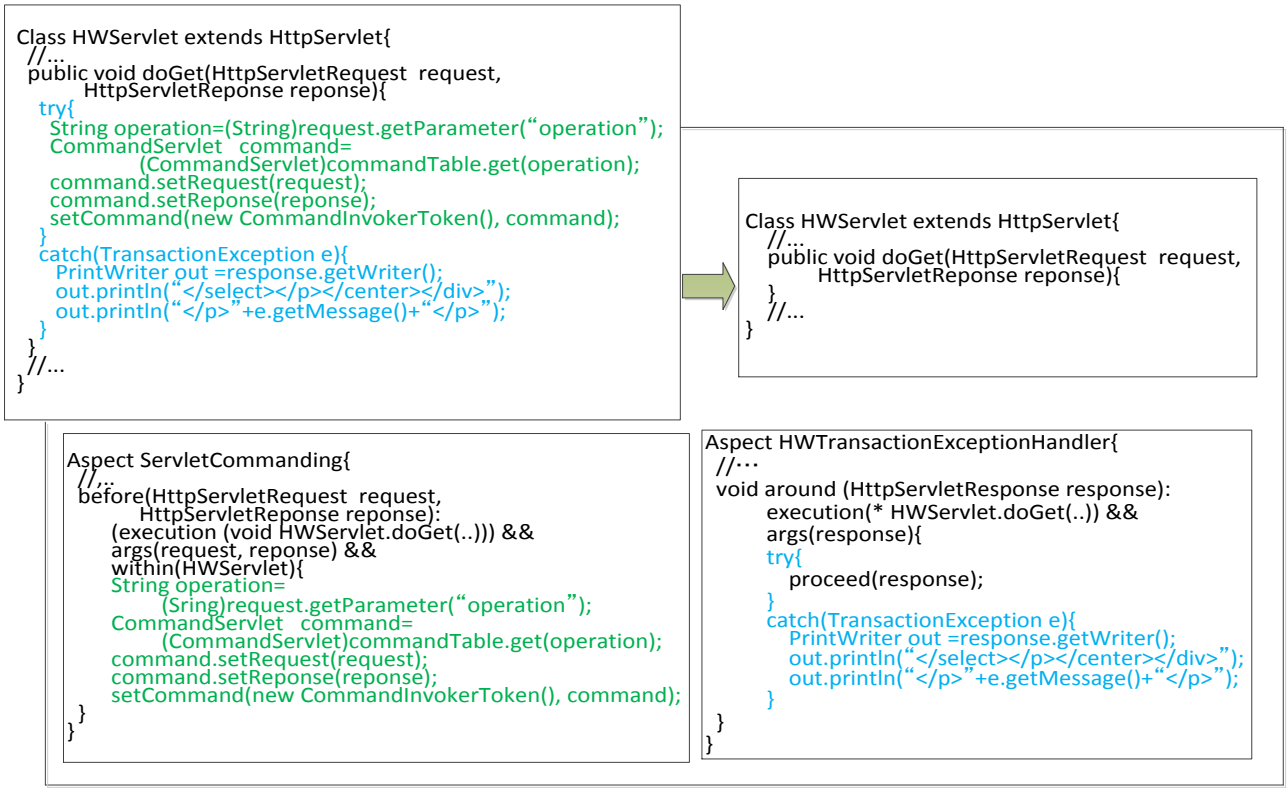

Figure 6.19: Example of refactoring: Exception Handler and Command Pattern

monitor precedes logging, and an incorrect execution time of search algorithm is returned. This case study allows us to verify aspect interference between two aspects which are implemented by system function.

In this section, we discuss the aspect interference in the aspect-oriented. In our preliminary solution, we expect to transform such aspect interference problem into a satisfiability problem. Such satisfiability problem can be resolved correctly and efficiently by the existing SMT solver. (1) If the SMT solver returns *unsat*, then it indicates that there is no aspect interference in the generating aspect-oriented program. (2) If the SMT solver returns *sat*, then it indicates that there is potential aspect interference in the generating aspect-oriented program.

Our idea successfully checks the aspect interference between aspects in three case studies. Our preliminary solution addresses this issue with following advantages: (1) we propose an automatic way for checking the aspect interference when refactoring cross-cutting concerns into aspects. It is difficult to verify the interference between aspects manually for the developer. The reason is that first the developer needs to understand the knowledge of aspect-oriented programming language and be able to find all the advice instructions which are advised at the same join point shadow. Second, the developer needs to analyze and understand the behavior of the program very well, since many details need to be handled. (2) Our solution is practical and efficient, because we transform such problem into satisfiability problem. In this way, we avoid to analyze and understand the semantic intention of the program, which is complicated and difficult to be implemented. (4) Our solution is applicability, because it can be easily implemented as a plugin or component in the existing tools, such as aspect refactoring tools and aspect interaction

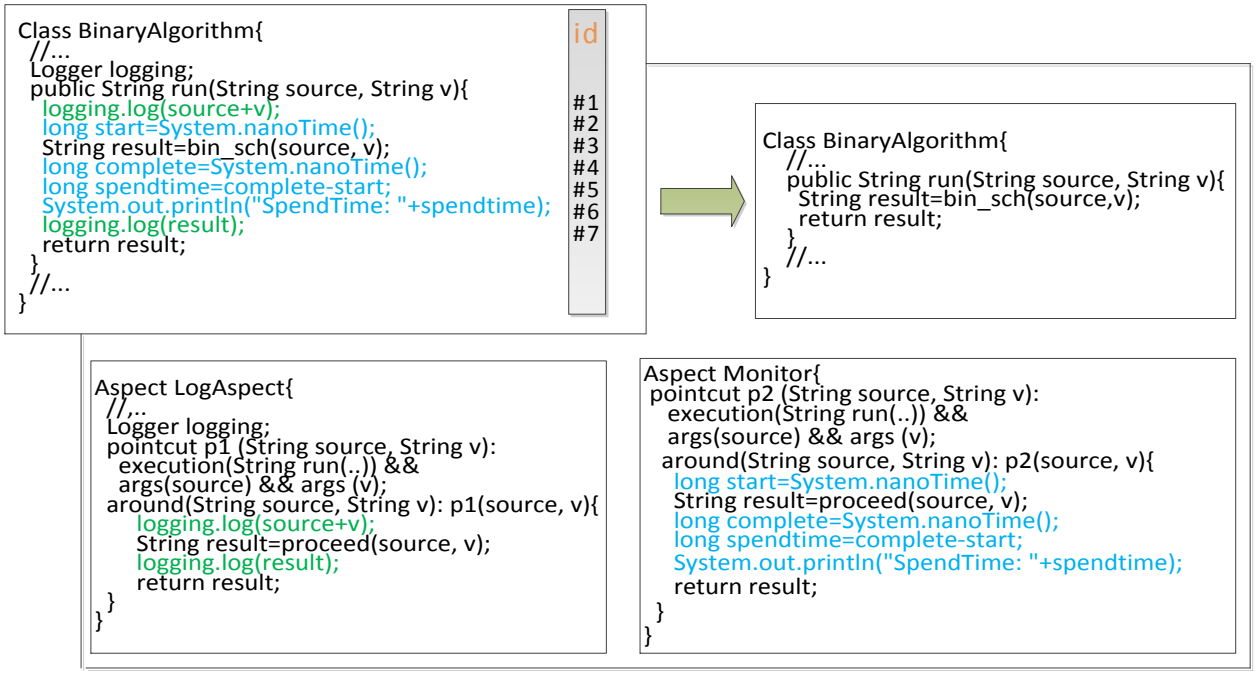

Figure 6.20: Example of refactoring: Logging and Monitoring

management tools.

However, we also find that in some rare special cases even the execution order of two dependent statements is changed, the behaviors of such two programs are still same. The counter example is shown as follows:

```
1  Class A{
2      int m=0;
3      void fun(){
4          m=1+2+3+4+5;              //id1
5          print("1+2+3+4+5=",m);    //id2
6          m=(1+5)*2+3;              //id3
7          print("1+2+3+4+5=",m);    //id4
8      }

10 }
```

In the above example, `id1` and `id3` are dependent. We find that when the execution order of `id1` and `id3` is changed, the behavior of the function `fun()` is not changed. The reason is that both of them are used to calculate the summation of values from one to five, but applies different ways. How to distinguish this situation when our solution gives awareness of potential errors of aspect interference is the subsequent work in the future.

Chapter 7

Related Work

In this Chapter, we first present some of the research work has been done on how to detect the crosscutting concerns from the object-oriented codes. Then we introduce some related work on management, documenting and detecting of aspect interactions. Finally, we present some previous works on dealing with fragile pointcut problems, involving some research that improved our solution in this dissertation.

7.1 Aspect Mining

Feat [75] creates a concern graph. Such concern is derived from the source code. A concern graph abstracts the implementation details of a concern by storing the key structure implementing a concern. The relationship between the different elements of a concern is documented by a concern graph. The developer is able to manipulate and navigate a concern representation at a more abstract level than the source code without investing any effort to create the abstract representation. Feat is able to iteratively create, visualize, and analyze concern graphs for Java program. The scattered concerns are able to be analyzed by the feat in an existing code base. By using the feat, however, the developer still need to map the relationship between the specific code and concern manually. It will spend a number of manual efforts. Our approach is based on the automatic aspect mining technique, which clustering the concern automatically.

Aspect Browser [76] is similar to the Feat as the visual editor tool of concern. By using the aspect browser, the developer can effectively describe non-hierarchical, interwoven, crosscutting module structures. The Aspect browser is different from ours in that our approach is based on the automatic aspect mining technique, and the aspect browser is a kind of semi-automatic tool.

Tonella et al. [9] propose an approach which utility formal concept analysis (FCA) to the execution trace in order to identify aspects. The feature location method based on formal concept analysis has been adapted to address the problem of aspect mining. Transformation of existing application to AOP is supported by their semi-automated aspect identification method, which requires the definition of use-case for the main application functionalities, when there are not already available. These use case scenario is a good indicator for crosscutting concerns. The use cases are used as elements when analyze the execution trace using FCA, and methods called from within use cases as properties. The concepts are considered as aspects if the methods belong to more than one class, or different methods from a same class occur in multiple use case specific concepts.

Another approach based on FCA is proposed by Tourwe [10]. They report an initial assessing the feasibility of formal concept analysis to discover aspectual view in the source code of an application automatically. They also define an aspectual view to represent the set of source code entities which are structurally somehow related to each other. These entities can be any source code artifact, such as a class hierarchy, a class, a method, a method parameter or an instance variable. The structural relation between these source code entities is arbitrary. An aspectual view can contain all source code entities that participate in visitor design patter. For example, aspectual view offer a view on the source code that is often crosscutting and that complements the standard views offered by traditional development environment. Hence, these views improve understandability and maintainability of the software. This research is different from our approach is that this approach identify the crosscutting concerns based on the specific indicator, and it cannot be changed for the different program. Our approach does not have a specific indicator. The input of optimized metrics should be different for the different program.

Serban et al. [14] propose an approach that is based k-means clustering algorithm. The input is a vector space model. The distance between the two methods is expressed using Euclidean distance. They propose four steps to identify crosscutting concerns in their approach. Step 1 is Computation, which computes the set of methods in the selected source code, and the attribute set values, for each method in the set. Step 2 is filtering, which eliminate some methods that belong to the some built-in classes like String, StringBuffer, etc. Step 3 is grouping, which groups the remaining set of methods into clusters. Step 4 is analysis, which analyzes the obtained clusters to discover which cluster contains methods belonging to crosscutting concerns. This approach used clustering based algorithm that as well as our approach. Nonetheless, our approach does not use only one indicator and one clustering algorithm. On the contrary, our approach uses a combination of metrics that each metric is considered as an indicator and compares two clustering algorithms. Another significant point in the difference is that our approach proposes an algorithm to optimize the combination of metrics.

Marin et al. [46]propose an aspect mining approach by using fan-in analysis. This approach aims at finding methods by computing the fan-in value metric for each method using static call graph of the system. They adopt a systematic approach which consists of several steps suitable for a high degree of automation. The technique has turned out to be flexible and easy to combine with other techniques such as clone detection or slicing. The key contributions of this research are the systematic approach used to identify the aspects. The fan-in analysis approach calculates the fan-in value of each method, filter accessor and auxiliary methods, and the number of considered methods is also limited by the fan-in threshold. However, our approach does not define any threshold to classify the crosscutting concerns. Our approach aims at using clustering algorithm to detect the crosscutting concerns by themselves. Fan-in value can be considered as one of metrics for the clustering algorithm.

7.2 Handling Fragile Pointcut Problem

There are several associated approaches which supporting analysis-based pointcuts or similar to our approach that attempt to overcome the fragile pointcut problem.

Expressive pointcut languages In order to make pointcut definitions less frag-

ile against program evolution, more expressive pointcut languages are currently under investigation. The Josh[96] and Alpha[97] are new AOP languages. Josh presents an AspectJ-like language. Such novel language supports extensible pointcut and a few mechanisms for generic description. Such extensible pointcut language is on the basis of the idea of open-compiler approach. This approach is not to make the source code of the compiler open to public. It is rather to develop easily understandable abstraction of the internal structure or behavior of the compiler and to provide the programming interface to customize the compiler through that abstraction. Josh allows expert developers to implement a new pointcut designator in Java, the developers can define a pointcut designator useful in particular application domain. Josh also allows a Java expression to be included within an inter-type declaration. This makes Josh avoiding redundant description of the inter-type declaration. However, Josh does not support the definition of a pointcut depending on other facets such as data flow, and Josh also does not support type checking and named pointcuts.

Alpha provides rich program information to the user defined pointcuts. The Alpha aspect language allows specifying pointcuts at a high-level of abstraction by providing different rich models of the program semantics and abstraction mechanisms analogous to functional abstraction. Alpha uses a logic programming language for the specification of pointcuts. Pointcut in Alpha are logic queries written in Prolog [98]. Nevertheless, the new pointcut language is difficult to be written by a developer who is not familiar with logic programming language. Furthermore, its dynamic execution model needs a complex compilation framework to achieve efficient performance.

Design Rules and XPI These approaches are to explicitly include the pointcut descriptions in the design of the software and to require software engineer to adhere to this design. The interfaces which are defined by the design rules, are implemented as Explicit Pointcut Interfaces (XPIs) using AspectJ [99]. XPIs define contracts or design rules that base code developers must observe. On the other hand, aspect developers must rely on the syntactic part XPIs to implement advices that do not directly reference source code elements. Pointcuts are declared globally and it is possible to verify constraints on the basis of XPIs. Our approach does not require defining strict design rules that the base code must conform to.

Annotations In order to avoid fragile pointcut problem, a number of research propose to define pointcuts in terms of explicit annotations in the code [100]. Similar to intentional view, annotations classify source code entities and thereby make explicit additional semantics that would otherwise be expressed through implicit programming conventions. However, using annotation addresses the fundamental cause of the problem only partially. While the pointcut definitions are defined in terms of semantic properties that would otherwise have remained implicit, the problem is displaced to the annotations themselves. Instead of requiring developers to adhere to implicit programming rules, we need them to annotate the base program explicitly. As a consequence, pointcuts are as brittle as the annotations to which they refer. When the base code has not been correctly annotated, or when annotations are not correctly updated when the base code evolves, the fragile pointcut problem still exists. Silva et al. [101] try to cope with this problem by propose maintainers to inform whether each target element in the lexical range of an annotation should be annotated or not. The central components of the proposed solution are annotator aspects that super impose annotations to the base code in a non-invasive way. This solution is particularly recommended in two situations: when it is possible to correlate

annotations, and, when it is possible to restrict the scope of an annotation.

Conceptual Models These pointcut languages propose that pointcuts should be defined in terms of conceptual models of the base program, instead of directly accessing the program abstract syntax tree. The rationale behind such approaches is that models are more robust to evolution, because they represent only abstract and consolidated concepts about the program domain. Kellens et al.[37] propose a novel pointcut construct language, namely model-based pointcuts. They use a more stable dependency replace the intimate dependency of point definitions on the base program. The model-based pointcuts are no longer defined in terms of how the program happens to be structured at a certain point in time. These new pointcuts are decoupled from the base program structure, fore conceptual model instead, so that this new kind of pointcut language is more powerful. However, similar to Alpha, this pointcut language is also difficult to be written by a developer who is not familiar with the new language. Our solution does not require the programmer to write the new pointcut by herself. Our framework generates the new pointcut automatically.

Pointcut Analysis These approaches tackle the fagile pointcut problem by analysing the program structure and pointcuts. SCoPE[102] is a compilation framework for AspectJ to support user-defined analysis-based pointcuts. Because it does not change AspectJ, it would be possible to integrate SCoPE and our framework to eliminate the runtime overhead of analysis-based pointcuts.

Anbalagan et al.[103] proposes a framework to generate pointcut mutants with different strengths, and to assist the developer inspect the pointcuts and their join points conveniently. The input of the framework is AspectJ source code and program bytecode. The output is ranked list of mutants and their join points. There are six components in this framework: pointcut parser, which identifies pointcuts in the given AspectJ source code; join point candidate identifier, which identifies join point candidates from the given Java bytecode, however there approach does not support to pointcuts related to dynamic contexts such as cflow; mutant generator, which generates pointcut mutants from the original pointcuts; pointcut tester, which verify the join point candidates compare with original pointcuts; mutant classifier, which classifies pointcut mutants into three types, strong pointcut mutant, wee pointcut mutant and neutral pointcut mutant; and distance measurer, which reduces the numbers of pointcut mutants according to the similar distance. However, this approach still depends on the name of the join points. The analysis-based pointcuts in our solution are not depending on the specific name.

Pointcut rejuvenation[104] and Pointcut-Doctor[105] are integration tools that assist developers to create and analyze the pointcuts. Pointcut rejuvenation picks up a lot of suggested join points which are ranked by the value of confidence after program evaluation. They use a path with some wildcard elements as a pattern to match the join points. The most striking difference is that the T-Pattern in our solution includes three kinds of intention property and the T-Pattern includes not only one path but also a set of rooted trees. Pointcut Doctor is extensions of AJDT tools that help developers create correct pointcuts by providing immediate diagnostic feedback. Their algorithm captures join points by changing the names in pointcut slightly. Pointcut Doctor could provide much useful information to help the programmer maintains the pointcuts. Our solution aims at generating a new pointcut (analysis-based pointcut) which does not need to be changed very often. In this case, the programmer does not need to spend much time maintaining the pointcuts.

7.3 Management of Aspect Interactions

To the best of our knowledge, this study is the first research which verifies the aspect interference in aspect-oriented refactoring. We summarize several related works in this section.

In terms of aspect-oriented modeling, despite a large of research [77, 78] have been few attempts to handle aspect interactions during modeling. A few of approaches have been to supply notations for explicitly documenting interactions, such as [67, 79]. Sanen et al. [67] classify four different types of aspect interaction: mutual exclusion (e.g, different policies implemented through different aspects), dependency (i.e., proper function of aspect requires composition with another specific aspect), and reinforcement and conflict representing positive and nefarious semantic interactions between aspects, respectively. They propose an approach for documenting aspect interactions based on these four types of interactions. Then think aspect interactions can be addressed if they are negative or kept as part of system documentation if they are positive. The form of knowledge about interaction which is proposed by Sannen can be useful at different states of the software like cycle. Bakre et al. [79] propose the Aspect Interaction Charts (AIC) that build on top of the Live Sequence Charts (LSC) [80] in order to capture the interactions among various aspects at join points. An AIC captures the interaction among various aspects and objects at a common join point in a graphical but formal manner. LSCs capture requirements in a formal specification. The play Engine enables the user to feed in scenarios using the play in mechanism and test them using the play out mechanism. However all of these approaches only focus on documenting aspect interactions.

Beyond the documenting, various research [81, 82, 83, 84, 85] propose an approach focus on analyzing conflicts and dependencies of aspect interactions based on modeling. In [81] the authors aim to manage interaction issues in an aspect-oriented middleware platform by allowing interaction contracts to be specified which then are enforced at runtime. Explicitly specifying these contracts improves the management and control of such interactions. Their work focuses on two main categories of aspect interactions: conflicts (two aspects being incompatible) and dependencies (one aspect requiring another). The solution in the paper includes a component model with a well-defined interaction model that supports a variety of relationships. These relationships are specified using interaction contracts that are evaluated at runtime to ensure conflicts do not occur and dependencies are fulfilled. The interaction model is based on shared elements (such as a common join point, a component instance or the base application). Its possible to specify both basic (requires and provides) and advanced (conflict, precedence and resolution) interaction contracts. The approach has been validated when they apply it to a series of interaction issues which occured when implementing services for CustAOWare (a flexible and customizable AO middleware platform). Similarly, Isabel et al. [83] propose an approach resolve the conflict by using the information which is appraised with respect to multi-criteria analysis.

Approaches in [82] and [84] are based on requirement modeling. Mehner et al. [82] propose an approach for analyzing aspect interactions at the level of requirements modeling. Their approach detect conflict and dependencies in behavioral specification of use cases refined with activity diagrams. Magno et al. [84] propose an approach that analyze the relevant concerns and scenarios to determine the composition order and rules. The analysis of concern interactions is based on AO requirements specification.

Shaker et al.[85] describe a formal language to detect interactions between aspects written in the Aspect-UML language. However it requires formal pre-condition and post-condition specification. All the above mentioned work manages aspect interactions based on requirement specification or design structure. None of them address the issue on detecting violation of aspect interaction from the program.

There are also several research that manage the interactions between aspects at the programming level. Blair et al. [86] proposed a two-level architecture that separates a feature's core behavior (in Java code) from aspect-orient code (in AspectJ). Their approach specifies how features interactions are resolved. However, they have not addressed the aspect interaction problem. Usage of the history of execution events as a basic mechanism for the definition of aspects is one way to handle aspect interactions, which has been proposed by several researchers, in particular [87, 88] and our previous work [89]. Our previous work gives an approach to detect the interactions between crosscutting concerns based on execution events in aspect refactoring. Taking advantages of semantic annotations to help detect interactions was also discussed by many researchers such as [90, 91, 92].

Barreiros et al. [91] define a novel concepts regarding aspect interaction management. The paper proposes some extensions to the AspectJ [5] language for detecting unintended aspect interactions. These extensions are aspect and advice cardinality, and meta-aspects. The authors start off by providing a classification of seven different types of aspect interactions. Some fundamental causes of undesired interactions also are discussed. Next, aspect and advice cardinality is defined to represent the absolute and relative proportions of aspect use and advice weaving. Aspect cardinality is the measure of the expected number of aspect bindings to an application while advice cardinality represents the expected number of advice weaving per aspect binding. Its the developers responsibility to ensure that multiple weavings at the same join point behave coherently depending on a certain applicable execution order. Finally, meta-aspects are generic, abstract specifications of concrete aspects with a number of advantages. These concrete aspects usually can be derived automatically with all generic pointcut definitions being instantiated into specific, narrow-scoped expressions.

Mussbacher et al. [92] present a novel approach for semantically detecting interactions between aspect-oriented scenarios. Each aspect is annotated with domain-specific markers. In addition, a separate influence model in their approach describes how semantic markers from different domains influence each other.

They propose three important extension to a framework [93]. They augment the underlying aspect language by introducing variables allowing the sharing of information between different parts of an aspect. The language becomes much more expressive but the absence of interactions between aspects can still be checked statically. They also propose some new composition operators for aspects. These operators are particularly useful to resolve conflicts between interacting aspects. Interaction arises when distinct aspects match the same join point. Each aspect is annotated with domain-specific markers and a separate influence model describes how semantic marker from different domains influences each other.

Static analysis is an another typical approach to detect interactions, such as [94, 95]. Stateful aspects [94] introduces a generic framework [93] for detecting aspect interactions on the basis of pointcuts with explicit states. They extend the underlying aspect language by introducing variables which allowing the sharing of information between different parts

of an aspect. The language becomes much more expressive but the absence of interactions between aspects can still be checked statically. They also propose some new composition operators for aspects. These operators are particularly useful to resolve conflicts between interacting aspects. Interaction arises when distinct aspects match the same join point. All in all, they propose an abstract formal semantics of their aspect language, which allow for detection of aspect interaction. This approach detects shared join points, however, Fraine et al. [95] propose an approach that has tried to go beyond shared join points to detect control flow-based interaction. They demonstrate the feasibility of a technique for managing control-flow interactions, one important kind of such interactions that they experience in layered architecture. This technique proposes to document aspects with policies that specify the expected interactions between different aspects, or between aspects and the base application. The policies are expressed as logic formulae that employ a set of predicates that represent relevant control-flow situation. They use the static analysis of the woven application to detect violations of these policies. Comparing with the previous research, our approach verifies the interference between aspects on the basis of SMT which does not need to extend aspect language to give any additional information.

Additionally, for the constraint solving, the SMT solver such Z3[106] and Yices[107] are becoming increasingly powerful with the advance of theorem provers and decision procedure. Said et al. [109] present a work that uses SMT-based analysis to detect data races. Our idea is inspired by this work. Unlike our approach, [109] does not consider control flow, they requires the whole trace read-write consistency.

Chapter 8

Conclusion and Future Work

In this dissertation, we have investigated the use of static program analysis technique in conjunction with the transformation of object-oriented program to aspect-oriented program approach to improve the accuracy of aspect mining result and translate the enumeration pointcut or name pattern-based pointcuts to analysis-based pointcuts. Our main thesis is that:

Aspect-oriented refactoring extends the benefit of aspect-oriented technique to legacy object-oriented programs without re-developing such legacy program. Static program analysis techniques can improve the aspect-oriented refactoring, and resolve the obstacle which existing in the refactoring process.

We have presented software metric selection and translation framework in this dissertation that resolves the significant problems which obstructed the aspect-oriented refactoring work.

Metrics selection proposed QAHSSS algorithm, a heuristic metric selection algorithm for clustering based aspect mining. This work addresses two issues. First, it reduced the size of the metric structure. Second, it removed the irrelevant metrics and increased the accuracy of aspect mining. Nataly is a framework for translating name-based pointcuts to analysis-based pointcuts. It addressed two issues: one is alleviating the fragile pointcut problem; the other is generating analysis-based pointcuts automatically instead of writing it manually. A deep discussion about aspect interference are also given in this dissertation. We have discussed how to extract the property and model from the source code and how to use SMT solver to check such models.

The results of our experiment evaluation lead us to conclude that the metrics selection approach would optimize the input metrics for clustering-based aspect mining algorithm, because the accuracy of aspect mining is improved after using the optimized metrics. The generated analysis-based pointcuts are more robust than the name pattern-based pointcuts or enumeration pointcuts, which are created in the aspect refactoring process.

The remaining parts of this chapter will recapitulate the technical contributions we developed and discuss a number of directions for future work.

8.1 Contributions

- We develop a heuristic algorithm QAHSSS for aspect mining. It would find the optimal subset of input metrics and remove irrelevant metrics from the given input set of metrics.

- We implement an IDE based tool for helping to detect the crosscutting concerns automatically. The tool supports two different clustering algorithms and 14 frequently used metrics in software quality engineer. The tool is also easy to extend for adding new metrics.

- Automatic inference of intention pattern: In Nataly, an intention pattern is abstracted from a set of relationship graphs of join points matched by a given input pointcut. The process of abstraction can be implemented automatically based on static program analysis.

- The automatic generated pointcuts are better integrated with the existing AOP language. In Nataly, the analysis-based pointcuts is implemented by conditional (if) pointcuts. It merely relies on existing AOP constructs and introspective reflection libraries. In addition, the existing compiler SCoPE [102] can be used for our analysis-based pointcuts directly.

- The automatic generated pointcuts have clear semantics. In Nataly, the analysis-based pointcut does not change the semantics of existing AOP language. This helps the programmer understand the program behavior after the translation.

- We pointcut an aspect interference problem in the aspect-oriented refactoring. The generating aspect-oriented program will be probably broken when such problem is not handled in the aspect-oriented refactoring.

- We propose an idea that transforms such aspect interference problem into a satisfiability problem. Furthermore, such satisfiability problem can be resolved correctly and efficiently by the existing SMT solver.

8.2 Future Work

To sum up, our plans for future work, We plan to extend our current work to support more aspect language in the future. For the metrics selection algorithm, classify and conclude what kinds of metrics are suitable for which kinds of projects for detecting crosscutting concerns is an interesting direction. In addition, using Intention Pattern to avoid advice repeated proceed problem is another interesting issue. We also plan to implement our idea of aspect interference awareness in the future. How to generate the CFG and ICFG automatically and how to find a way to extract the constraint property and model automatically will be the continue work in the future. Last but not least, how to implement our approaches for various legacy object-oriented program in practical is a significant direction. How to integrate all the techniques proposed in the aspect-oriented refactoring research to an integrated tool to help software engineers reusing their legacy object-oriented software system with the aspect-oriented techniques easier in practical is another attractive direction.

Bibliography

[1] Parnas D. L., "On the criteria to be used in decomposing systems into modules", Communications of the ACM 15 (12), pp. 1053-1059, December (1972).

[2] Dijsktra E., "A Discipline of Programming", Prentice Hall, (1976).

[3] O. G. Imed Hammouda, Olcay Guldogan. "Tool-supported customization of uml class diagrams for learning complex system models", In Proceedings of the 12th IEEE International Workshop on Program Comprehension, page 24. IEEE Computer Society, (2004).

[4] J. Aldrich. "Evaluating module systems for crosscutting concerns", In University of Washington, (2000).

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. "An overview of aspectj", In Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP '01, pp. 327-353. Springer-Verlag, (2001).

[6] Gosling J., Joy B., Steele G., "The Java Language Specification", second edition. Addison Wesley, (1996).

[7] D. Binkley, M. M. Ceccato, M. Harman, F. Ricca and P. Tonella. "Tool-Supported Refactoring of Existing Object-Oriented Code into Aspects, IEEE Transactions on Software Engineering, vol.32, pp.698-717, (2006).

[8] L. Cole and P. Borba. "Deriving refactorings for AspectJ", In Proceedings of the 4th international conference on Aspect-oriented software development (AOSD '05), pp. 123-134, (2005).

[9] P. Tonella and M. Ceccato, Aspect Mining through the Formal Concept Analysis of Execution Traces, Proc. 11th Working Conf. Reverse Eng. (WCRE), pp. 112-121, (2004).

[10] T. Tourwe and K. Mens, Mining Aspectual Views Using Formal Concept Analysis, Proc. Fourth IEEE Intl Workshop Source Code Analysis and Manipulation (SCAM 04), pp. 97-106, (2004).

[11] S. Hanenberg, C. Oberschulte and R. Unland, Refactoring of Aspect-Oriented Software, Proc. Fourth Ann. Int'l Conf. Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays), pp. 19-35, (2003).

[12] M.P. Monteiro and J.M. Fernandes, Towards a Catalog of Aspect-Oriented Refactorings, Proc. Fourth Int'l Conf. Aspect-Oriented Software Development (AOSD), pp. 111-122, (2005).

[13] A. van Deursen, M. Marin and L. Moonen, Aspect Mining and Refactoring, Proc. First Int'l Workshop Refactoring: Achievements, Challenges, Effects (REFACE), with WCRE, (2003).

[14] G. Serban and G. Soffa Moldovan. "A new k-means based clustering algorithm in aspect mining", In Proceedings of 8th International Symposium on Symbolic and Romania, (2006).

[15] F.J. Mitropoulos S.G. Maisikeli. "Aspect mining using self-organizing maps with method level dynamic software metrics as input vectors", In Proceedings of 2nd International Conference on Software Technology and Engineering, (2010).

[16] G. Yao Z. Danfeng and C. Xiangqun. Automated aspect recommendation through clustering-based fan-in analysis. In Proceedings of 23rd IEEE/ACM International Conference on Automated Software Engineering, (2008).

[17] G. Serban G.S. Moldovan. "Aspect mining using a vector-space model based clustering approach", In Proceedings of Linking Aspect Technology and Evolution workshop, (2006).

[18] M. Kamber J. Han. "Data Mining: Concepts and Techniques", Morgan Kaufmann Publisher, (2001).

[19] H. Liu M. Dash. "Feature selection for classification", Journal of Intelligent Data Analysis, (1997).

[20] G. Xu and A. Rountev. "AJANA: a general framework for source-code level interprocedural dataflow analysis of AspectJ software", In Proceedings of the 7th international conference on Aspect-oriented software development (AOSD 08). pp.36-47, (2008).

[21] F. Nielson, H. R. Nielson, and C. Hankin, "Principles of Program Analysis", 2nd. Berlin, Germany: Springer, (2005).

[22] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers: principles, techniques, and tools", Addison-Wesley, (1986).

[23] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers, ACM SIGOPS Operating Systems Review, vol. 40, (2006).

[24] C. Baier and J. Katoen, "Principles of model checking", The MIT Press, (2008).

[25] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes,J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming, in Proceedings of the European Conference on Object-Oriented Programming, ECOOP97. Springer, pp. 220242, (1997).

[26] Hillsdale E., Hugunin J. "Advice Weaving in AspectJ", Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2003), pp. 26-35, (2004).

[27] Avgustinov P., Christensen A. S., Hendren L., Kuzins S., Lhotk J., de Moor O., Sereni D., Sittampalam G., Tibble J., "abc: An Extensible AspectJ Compiler", In Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005), pp. 87-98, (2005).

[28] Popovici A., Alonso G., Gross T., "Just-In-Time Aspects: Efficient Dynamic Weaving for Java", In Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003), pp. 100-109, (2003).

[29] Lopes C. V., D, "A Language Framework for Distributed Computing", Ph.D. thesis, College of Computer Science, Northeastern University, (1997).

[30] Lopes C. V., Kiczales G., D: "A Language Framework for Distributed Programming", Xerox PARC, Palo Alto, CA. Technical report SPL97-010 P9710047, (1997).

[31] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. G., "Getting Started with AspectJ", Communications of the ACM, 44(10):59-65, (2001).

[32] Laddad R., "AspectJ in Action. Practical Aspect-Oriented Programming", Manning (2003).

[33] Kersten M., "AOP tools comparison, parts 1 and 2". Developer Works (first article of the AOP@Work series), IBM, (2005).

[34] Filman R. E., Friedman D. P., "Aspect-Oriented Programming is Quantification and Obliviousness", Workshop on Advanced Separation of Concerns at OOPSLA 2000, Minneapolis, (2000).

[35] Stoerzer, M., Graf, J. "Using pointcut delta analysis to support evolution of aspect-oriented software", In 21st IEEE International Conference on Software Maintenance (ICSM), (2005).

[36] Koppen, C., Stoerzer,M. "Pcdiff: Attacking the fragile pointcut problem", In First European Interactive Workshop on Aspects in Software (EIWAS), (2004).

[37] A. Kellens, K. Mens, J. Brichau, and K. Gybels. "Managing the evolution of aspect-oriented software with model-based pointcuts", In Proceedings of the 20th European conference on Object-Oriented Programming (ECOOP'06), (2006).

[38] Breu, S., Krinke, J. "Aspect mining using event traces", In Conference on Automated Software Engineering (ASE 04), (2004).

[39] Breu, S. "Towards hybrid aspect mining: Static extensions to dynamic aspect mining", In 1st Workshop on Aspect Reverse Engineering, (2004).

[40] Ganter, B., Wille, R. "Formal Concept Analysis: Mathematical Foundations", Spring-Verlag, (1999).

[41] Shepherd, D., Tourwe, T., Pollock, L. "Using language clues to discover crosscutting concerns", In Workshop on the Modeling and Analysis of Concerns, (2005).

[42] Gybels, K., kellens, A. "An experiment in using inductive logic programming to uncover pointcuts", In First European Interactive Workshop on Aspects in Software, (2004).

[43] Gybels, K., kellens, A. "Experiences with identifying aspects in Smalltalk using unique methods", In Workshop on Linking Aspect Technology and Evolution, (2005).

[44] J. Han. "Data Mining: Concepts and Techniques", Morgan Kaufmann Publishers Inc., (2005).

[45] Karanjkar, S. "Development of graph clustering algorithms", Master's thesis, University of Minnesota, (1998).

[46] Marin, M., van Deursen, A., Moonen, L. "Identifying aspects using fan-in analysis", In Working Conference on Reverse Engineering (WCRE 04), IEEE Computer Society, pp. 132-141, (2004).

[47] D. Shepherd, E. Gibson, and L. Pollock. "Design and evaluation of an automated aspect mining tool", In International Conference on Software Engineering Research and Practice, (2004).

[48] M. Bruntink. "Aspect mining using clone class metrics", In 1st Workshop on Aspect Reverse Engineering, (2004).

[49] M. Bruntink, A.v.Deursen, R.v. Engelen, and T.Toruwe. "An evaluation of clone detection techniques for identifying crosscutting concerns", In Proceeding of IEEE International Conferences on Software Maintenance (ICSM 04). IEEE Computer Society Press, (2004).

[50] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. "Refactoring: Improving the Design of Existing Code", Addison-Wesley, (1999).

[51] Garcia A., SantAnna C., Figueiredo E., Kulesza U., Lucena C., Staa A., "Modularizing Design Patterns with Aspects: A Quantitative Study", In Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005), ACM press, pp. 3-14, (2005).

[52] Ran E., and Mathieu V,. "Untangling: a slice extraction refactoring", In AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development, pages 93-101, (2004).

[53] Jan H., Gail C. Murphy, and Gregor K. "Role-based refactoring of crosscutting concerns", In AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, pp. 135-146, (2005).

[54] Morris, J., hirst, G. "Lexical cohesion computed by thesaural relations as an indicator of the structure of text", Computational Linguistics, pp. 21-48, (1991).

[55] Sable Benchmarks. The classic figure editor example. Available from: http://www.sable.mcgill.ca/benchmarks/#aspectj.

[56] Peter J. Rousseeuw Leonard Kaufman. "Finding Groups in Data: An Introduction to Cluster Analysis", Wiley-Interscience, (1990).

[57] F.J. Mitropoulos S.G. Maisikeli. "Aspect mining using self-organizing maps with method level dynamic software metrics as input vectors", In Proceedings of 2nd International Conference on Software Technology and Engineering, (2010).

[58] G. Serban G.S. Moldovan. "Aspect mining using a vector-space model based clustering approach", In Proceedings of Linking Aspect Technology and Evolution workshop, (2006).

[59] T. Kohonen. "Self-Organizing maps", Series in Information Sciences, (1997).

[60] E. Alhoniemi J. Vesanto, J. Himberg and J. Parhankangas. "Self-organizing map in matlab: the som toolbox", In Proceedings of the Matlab DSP Conference, (1999).

[61] K. Kiviluoto. "Topology preservation in self-organizing maps", In Proceedings of International Conference on Neural Networks, (1996).

[62] J. Whitehead E. Sunghun. "When functions change their names: Automatic detection of origin relationships", In Proceedings of 12th Working Conference on Reverse Engineering (WCRE'05), (2005).

[63] L. Blair, G. Blair, and J. Pang. "Feature interaction outside a telecom domain", Workshop on Feature Interaction in Composed Systems, (2001).

[64] X. Liu, G. Huang, W. Zhang, and H. Mei. "Feature interaction problems in middleware services", International Conference on Feature Interactions (ICFI '05), (2005).

[65] R. E. Filman, T. Elrad, S. Clarke, and M. Aksit. "Aspect-oriented software development", Addison-Wesley, (2005).

[66] R.E. Filman, K.Havelund, and K.Technology. Realizing aspects by transforming for events. In IEEE, editor, Automated Software Engineering, ASE '02, (2002).

[67] Frans Sanen , Eddy Truyen , Wouter Joosen. "Classifying And Documenting Aspect Interactions", In Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software, (2006).

[68] S. Sandra I. Casas, J. J. Baltasar G. P., and C. Claudia A. Marcos. "MEDIATOR: an AOP Tool to Support Conflicts among Aspects", International Journal of Software Engineering and Its Applications (IJSEIA), pp. 33-44, (2009).

[69] G. Xu and A. Rountev. "Regression test selection for AspectJ software", In Proceeding of International conference on Software Engineering (ICSE '07), pp. 65-74, (2007).

[70] K. Gybels and A. Kellens, "An Experiment in Using Inductive Logic Programming to Uncover Pointcuts", Proc. First European Interactive Workshop Aspects in Software, (2004).

[71] C.Laffra. Dijkstra's shortest path algorithm. Available from: http://carbon.cudenver.edu/hgreenbe/courses/dijkstra/Dijkstrapplet.html

[72] Stephen H. Kan. "Metrics and Models in Software Quality Engineering", Addison-Wesley Longman Publishing Co., Inc., (2002).

[73] G. Yao Z. Danfeng and C. Xiangqun. "Automated aspect recommendation through clustering-based fan-in analysis", In Proceedings of 23rd IEEE/ACM International Conference on Automated Software Engineering, (2008).

[74] Stephen H. Kan. "Metrics and Models in Software Quality Engineering", Addison-Wesley Longman Publishing Co., Inc., (2002).

[75] Martin P. Robillard and Gail C. Murphy. "Concern graphs: finding and describing concerns using structural program dependencies", In Proceedings of the 24th International Conference on Software Engineering. ICSE, (2002).

[76] J. Neddenriep M. Shonle and W. Griswold. "Aspectbrowser for eclipse: a case study in plug-in retargeting", In Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange, (2004).

[77] Klein J, Helouet L, Jezequel J-M. "Semantic-based weaving of scenarios", Aspect-Oriented Software Development ACM Press, pp. 2738. (2006).

[78] France R, Ray I, Georg G, Ghosh S. "Aspect-oriented approach to early design modeling", Software, IEE Proceedings, vol. 151, pp. 173-185. (2004).

[79] Shubhanan Bakre and Tzilla Elrad. "Scenario based resolution of aspect interactions with aspect interaction charts", In Proceedings of the 10th international workshop on Aspect-oriented modeling (AOM '07), pp. 1-6, (2007).

[80] Harel and R. Marelly. "Come, Lets Play:Scenario-Based Programming Using LSCs and the Play-Engine", Springer-Verlag, (2003).

[81] P. Greenwood, B. Lagaisse, F. Sanen, G. Coulson, A. Rashid, E. Truyen and W. Joosen. "Interactions in Aspect-Oriented Middleware", In proceedings of 2nd International Workshop on Aspects, Dependencies and Interactions at ECOOP '07, (2007).

[82] Mehner, K.; Monga, M.; Taentzer, G., "Interaction Analysis in Aspect-Oriented Models", Requirements Engineering, 14th IEEE International Conference, pp.69-78, (2006).

[83] Isabel S. B., Filipe V., Ana M., and Rita A.R. "Handling conflicts in aspectual requirements compositions", In Transactions on aspect-oriented software development III, Awais Rashid and Mehmet Aksit Springer-Verlag, (2007).

[84] Magno, J., Moreira, A. "Concern Interactions and Tradeoffs: Preparing Requirements to Architecture", Aspects, Dependencies and Interactions Workshop at ECOOP '06, (2006).

[85] Shaker, P., Peters, D., "Design-level Detection of Interactions in Aspect-Oriented Systems", Aspects, Dependencies and Interactions Workshop at ECOOP '06, (2006).

[86] Blair L, Pang J. "Aspect-oriented solutions to feature interaction concerns using AspectJ", Feature interactions in telecommunications and software systems VII, IOS Press, pp 87-104, (2003).

[87] Robert E. F., Klaus H. "Realizing aspects by transforming for events", In IEEE, editor, Automated Software Engineering (ASE '02), (2002).

[88] Robert J. Walker , Gail C. Murphy. "Joinpoints as ordered events: Towards applying implicit context to aspect-orientation", In Proceedings of ASOC Workshop at ICSE '01, (2001).

[89] Lin W., Tomoyuki A., Masato S. "Interaction awareness for aspect refactoring", In Proceedings of the 8th international workshop on Advanced modularization techniques at AOSD '13. (2013).

[90] Bergmans LMJ. "Towards detection of semantic conflicts between crosscutting concerns", Analysis of aspect-oriented software (AAOS) at ECOOP '03, (2003).

[91] Barreiros, J., Moreira, A. "Aspect interaction management with meta-aspects and advice cardinality", In Proceedings of the Second International Workshop on Aspects, Dependencies and Interactions at ECOOP '07, pp. 11-16. (2007).

[92] Mussbacher, G.; Whittle, J.; Amyot, Daniel, "Semantic-Based Interaction Detection in Aspect-Oriented Scenarios", 17th IEEE International Conference on Requirements Engineering (RE '09), pp.203-212, 2009.

[93] Douence, P. Fradet, and M. Siidholt. "A framework for the detection and resolution of aspect interactions", In Proc. of the Conf. on Generative Programming and Component Engineering, pages 173-188, (2002).

[94] Douence, R., Fradet, P., Sudholt, M. "Composition, reuse, and interaction analysis of stateful aspects", In Proceedings of the 3rd international Conference of Aspect-oriented Software Development (AOSD '04), (2004).

[95] de Fraine B, Quiroga PD, Jonckers V. "Management of aspect interactions using statically verified control flow relations", In: Workshop on aspects, dependencies and interactions at ECOOP '08, (2008).

[96] S. Chiba and K. Nakagawa. "Josh: An open AspectJ-like language", In Proceeding of the 3th International Conference on Aspect-Oriented Software Development, pp. 102-111, (2004).

[97] K. Ostermann, M. Mezini, and C. Bockisch. "Expressive Pointcuts for increased modularity", In Proceeding of the 19th European Conference on Object-Oriented Programming, pp. 214-240, (2005).

[98] L. Sterling and E. Shapiro. "The Art of Prolog", MIT Press, (1994).

[99] W.G. Griswold, K.J. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. "Modular software design with crosscutting interfaces", IEEE Software, pp. 51-60, (2006).

[100] Kiczales, G., Mezini, M. "Separation of concerns with procedures, annotations, advice and pointcuts", In Proceedings of European Conference on Object-Oriented Programming (ECOOP), (2005).

[101] L. Silva, S. Domingues, and M. Valente. "Non-invasive and non-scattered annotations for more robust pointcuts", In Proceeding of 24th International Conference on Software Maintenance, pp. 67-76, (2008).

[102] T. Aotani and H. Masuhara. "SCoPE: An AspectJ compiler for supporting user-defined analysis-based pointcuts", In Proceeding of the 6th International Conference on Aspect-Oriented Software Development, pp. 161-172, (2007).

[103] P. Anbalagan and T. Xie. "Automated generation of pointcut mutants for testing pointcuts in AspectJ programs", In Proceeding of 19th International Symposium on Software Reliability Engineering, pp. 239-248, (2008).

[104] R. Khatchadourian, P. Greenwood, A. Rashid, and G. Xu. "Pointcut rejuvenation: Recovering pointcut expression in evolving aspect-oreient software", In Proceeding of 24th International Conference on Automated Software Engineering, pp. 575-579, (2009).

[105] L. Ye and K. De Volder. "Tool support for understanding and diagnosing pointcut expressions", In Proceeding of the 7th International Conference on Aspect-Oriented Software Development, pp. 144-155, (2008).

[106] Leonardo De Moura and Nikolaj Bjorner. Z3: an efficient SMT solver in TACAS, (2008).

[107] Bruno Dutertre and Leonardo De moura. The Yices SMT solver. Technical report, (2006).

[108] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Saisfiability Modulo Theories. volume 185, chapter 26, pages 825-885. IOS Press, (2009).

[109] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. "Generating data race witnesses by an SMT-based analysis", In Proceedings of the Third international conference on NASA Formal methods (NFM'11),pp. 313-327, (2011).

Publications

[1] Lin Wang, Tomoyuki Aotani, Automatic translation from name-based pointcuts to analysis-based pointcuts for robust aspects, on 8th Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'11) in TOOLS Federated Conferences. ACM, 2011.

[2] Lin Wang, Tomoyuki Aotani, and Masato Suzuki. Feature selection for clustering based aspect mining. In Proceedings of the 4th international workshop on Variability & composition(VariComp '13) in AOSD13. pp. 7-12. ACM, 2013.

[3] Lin Wang, Tomoyuki Aotani, and Masato Suzuki. Interaction awareness for aspect refactoring. In Proceedings of the 8th international workshop on Advanced modularization techniques (AOAsia '13) in AOSD13. ACM, pp. 15-18. 2013.

[4] Lin Wang, Tomoyuki Aotani, and Masato Suzuki. Improving the quality of AspectJ application Translating name-based pointcuts to analysis-based pointcuts. In Proceeding of the 14th International Conference on quality Software (QSIC14). pp. 27-36. IEEE, 2014.