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

By Lin Wang

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Koichiro Ochimizu (JAIST)
Professor ZhiYong Feng (TU)

September, 2011

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

By Lin Wang (0910216)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Koichiro Ochimizu (JAIST)
Professor ZhiYong Feng (TU)

and approved by
Professor Koichiro Ochimizu (JAIST)
Associate Professor Masato Suzuki (JAIST)
Associate Professor Toshiaki Aoki (JAIST)
Professor ZhaoPeng Meng (TU)
Associate Professor YiKui Zhang (TU)

August, 2011 (Submitted)

Abstract

Separation of concerns(SoC) is an important principle in software engineering. Without it large software system simply could not be realized. Aspect-Oriented programming(AOP) improves SoC by modularizing crosscutting concerns. Unfortunately todays mainstream AOP languages suffer from fragile pointcut problem. They are fragile because they break easily if the names of the methods or classes are changed when program evolved. We compared several important researches which attempt to solve pointcut fragility, and observed that new pointcut languages are very different from original one, so they are difficult to be written by a programmer.

In this dissertation we propose a framework called Nataly, which translates name-based pointcuts into analysis-based pointcuts automatically. Our approach can not only alleviate pointcut fragile problem but also bridge a gap between original name-based pointcut and other robust one. Name-based pointcuts directly use class and method names; they merely check that a called/executed method has the specified name/type. Analysis-based pointcuts are proposed as an approach to overcome the fragility. They use static program analysis rather than names, and match the join points that satisfy the match strategy checked by the analysis. One of the problems in using the analysis-based pointcuts is difficulty in implementing correct program analysis. We tackle the problem by translations from name-based pointcuts to analysis-based ones. We implemented the Nataly framework in Java. We also illustrate a case study that evaluates robustness of name-based pointcuts and analysis-based pointcuts in face of seven possible change scenarios to the classic Figure Editor System.

Acknowledgments

First and foremost, I would like to show my deepest gratitude to my supervisor - Professor Koichiro Ochimizu, for his guidance through the course of study in JAIST, Japan. His invaluable advices and support have carried me through difficulties and joy during my study in Japan. He has made me interested in science and doing research. My study could not have been completed without him.

Secondly, I would like to express my sincere thanks to Assistant Professor Tomoyuki Aotani, for his constant guidance, advice, assistance and support during my whole Master's program. Without his advices and support, I could not have made a good progress.

Thirdly, I would like to thank Associate Professor Masato Suzuki, for his encouragement and helpful comments.

Fourthly, I would like to thank Professor ZhiYong Feng, for his guidance, advice and support in TianJin University.

I am grateful to thanks all members in our laboratory, for their kindly helps to me not only in the research but also in the daily life.

Furthermore, I would like to acknowledge TianJin University, for their support in completing dual degree program and first year of this course in CHINA.

Special thanks to Xiaonan Shi for her helps and supports in JAIST.

Finally and most importantly, I would like to thank my family for encouraging and supporting me during my whole study in Japan, and Xuan Li for her unconditional love, support and understanding.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern software projects are of large scale, often involving tens of thousands of days of work efforts, and millions of lines of code. Moreover our limited minds cannot possibly consider everything and solve everything at once. Therefore, the software is too complex to design merely in a single view, and it is necessary to separate the different concerns in a large piece of software into smaller, more manageable units. Separation of concerns is one of the key principles in software engineering, and it refers to the ability to separate programs into distinct features. Effective separation of concerns makes a program easier to understand, change and debug[4]. Concerns are used to organize and decompose software into manageable and comprehensible parts[14]. Modularity or modular programming is a way of achieving separation of concerns. Object-oriented programming (OOP)[20] is a way of modularizing common concerns, classes in object-oriented model perform a single specific function. In many case, we find that many parts of our system have code fragments for logging, persistence, debugging, authorization, tracing, exception handling, and other such tasks. As a result, they have to be coordinated with other functional unites and code scattered usually throughout several functional units exist in their concerns. Thus, OOP does not do as a good job in these situations. Aspect-Oriented programming (AOP)[18] fills this void. In contrast with OOP, AOP is a way of modularizing crosscutting concerns. Crosscutting concern is a concern that affects several classes or modules. The modules for the crosscutting concerns are called aspects, aspects encapsulate behaviors that affect multiple classes into reusable modules. AspectJ[17] is the most prevalent and studied in mainstream aspect languages. It is a practical aspect-oriented to Java. AspectJ introduces a new concept, join points, and a few new programming constructs, such as pointcuts and advice. A pointcut is a construct designed to identify and select join points. Advice defines code to be executed when a join point is reached. Pointcuts match join points, which are well defined points in the execution of a program, so it is a key element in aspect-oriented languages.

Using names and wildcards to capture join points over programs is the most basic way to define the pointcuts. As the program evolves, changes in the structure or naming conventions can make advice accidentally execute an unwanted advice or miss execute some required ones. In other words, there is tight coupling between aspect and structure

of base codw. In this case, when the program evolved, as even a small modification to the program, the pointcut itself have to evolve along with the base-code. This problem is called fragile pointcut problem[19]. Hence it is important to define pointcuts in a robust way against program evolution. Robustness means the pointcuts are able to continue to capture the correct set of join points without alters its form in future version of the base-code.

There are many approaches[5, 6, 8, 9, 13, 15, 16, 22, 23, 24, 25] that attempt to overcome pointcuts' fragility. These approaches propose several new pointcut language constructs to improve pointcuts' expressiveness. There is, however, no automatic way to translate name-based pointcuts into robust ones, and hence programmers have to rewrite the existing pointcuts manually. Unfortunately, this is often difficult because new constructs are very different from the original languages. For example [21] uses a logic programmig language for the specification of pointcuts, it is very different from the name-based pointcuts.

This dissertation reports our work on develop a framework to translate name-based pointcuts into analysis-based pointcuts automatically. Analysis-based pointcut is defined by using static program analysis. There are three significant components in our framework, namely, relation analyzer, pattern generator and code generator. Given base-code aspects, it first analyzes the program to extract relation maps that represent relationships among fields, methods, classes and interfaces. Then these relation maps are used by pattern generator to build relationship trees which correspond to the given pointcut. Next extract intention pattern from the relationship trees for each pointcut to be translated. Finally, the code generator generates analysis-based pointcuts by using the pattern. In addition we recommend programmer use stable interfaces/names to define their target relevant to particular concern, for it can strength our intention pattern, and increase the accuracy of capture correct set of join points. We implement the framework for AspectJ in Java, and show that the generated analysis-based pointcuts are more robust than their name-based counterparts under various program changes, but still fragile under complicate changes, through a case study.

The outline of this paper is organized as follows: Chapter 2 contains a background on the field of Crosscutting concerns, Aspect-oriented programming and several Aspect-oriented programming languages. Together with analysis-based pointcut which applied in our approach, and some other researches related to the approach of overcome fragile pointcut problem are compared and analyzed in this chapter. Chapter 3 illustrate our framework, and explain how can it alleviate the fragility. Chapter 4 illustrates a case study to evaluate robustness of our approach. Chapter 5 draws the overall conclusion of this research and our plans for future work.

Chapter 2

Setting the Scene

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

Section 2.1 introduces the research background, which covers work in different research domains that our dissertation draw on, including crosscutting concerns, aspect-oriented programming, AspectJ and other AOP languages and fragile pointcut program. Section 2.2 presents Analysis-based Pointcut[6] and related works. Analysis-based pointcut is one of the approaches to overcome the fragility. A piece of code as an example is also demonstrated in this section. By comparing these approaches, we observe a gap between traditional pointcut language and a new robust pointcut language, and then propose our approach that attempt to generate analysis-based pointcuts instead of traditional name-based pointcuts automatically to fill the gap.

2.1 Research Background

2.1.1 Crosscutting concerns

Object-Oriented Programming (OOP)[20] is the most popular programming paradigm today. The evolution from machine level language to OOP reflects that readability and reusability in designing software are more and more significant. Object-oriented programming language is a way of modularizing common concerns. Classes in object-oriented model perform a single specific function, however, not all concerns can be encapsulated properly in a functional decomposition, and these classes often share common code fragments. For example, tracing and logging are concerns that have common code fragment that are usually distinct from the functional units. As a result, they have to be coordinated with other functional units and code scattered usually throughout several functional units exist in their concerns. Therefore, Object-oriented programming does not do as a good job in these situations. We call these concerns as crosscutting concerns. Crosscutting concerns are computational units of a program that provide similar functionalities, however cannot be abstracted into a standalone module because of the limitations of the programming language. Crosscutting concerns make the code unclean and unmodularized. Suppose

we change in the crosscutting concerns, meanwhile, we have to change code in several different places. Therefore it leads to code scattering and tangling.

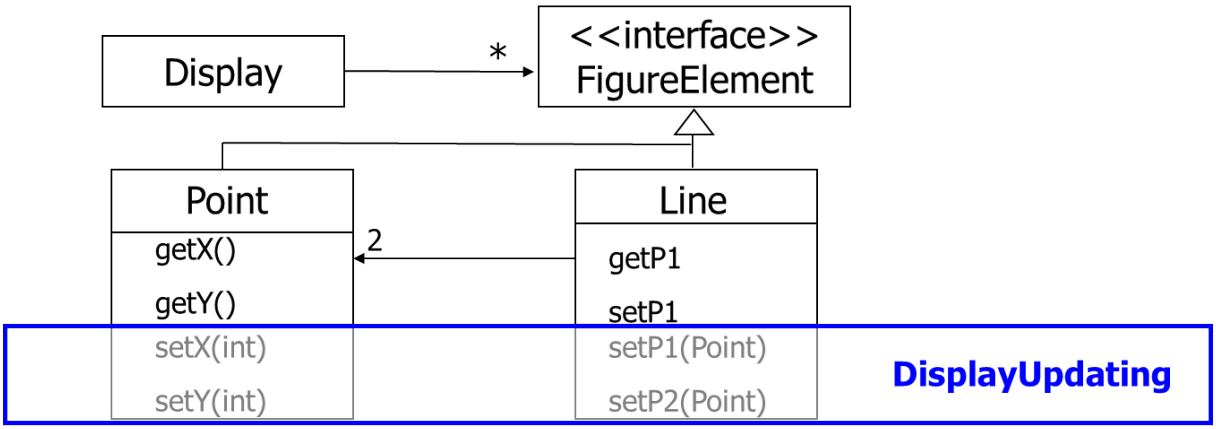

Figure 2.1: Example of crosscutting concerns in figure editor system.

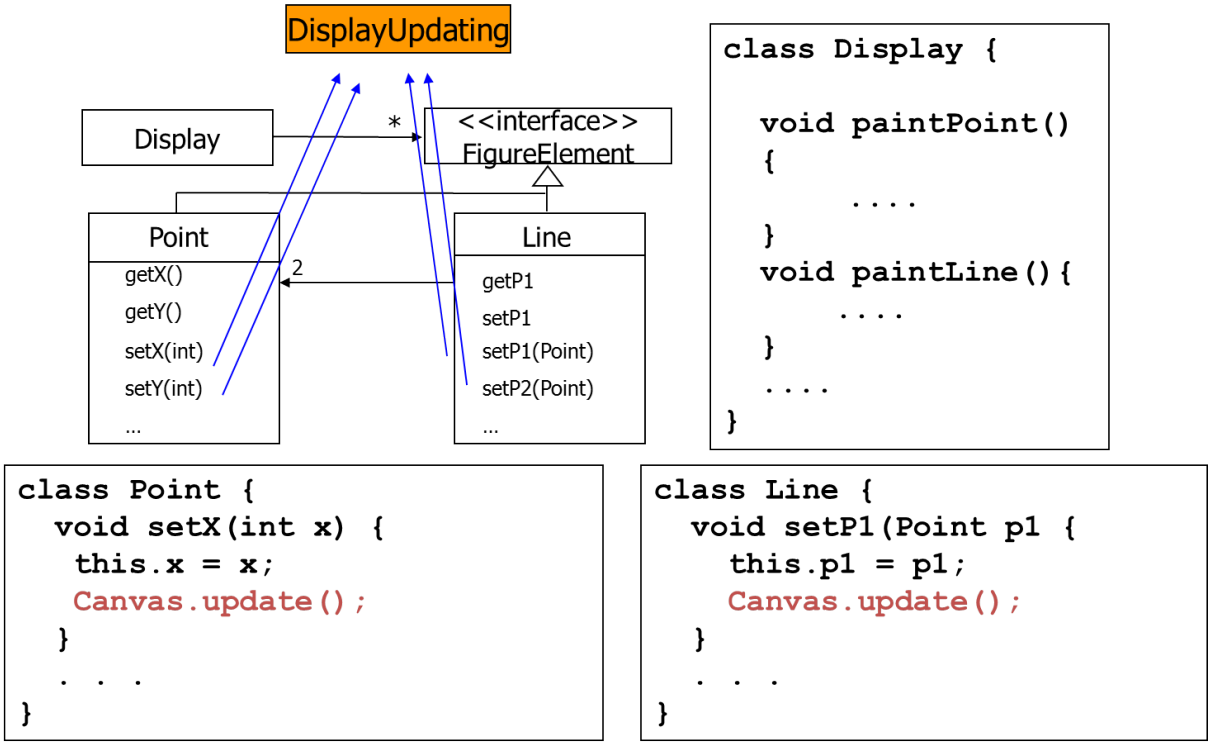

Figure 2.2: Implementtion of crosscutting concerns in figure editor system without AOP

EXAMPLE 2.1. *A simplicity of crosscutting concerns example is shown in Figure 2.1. The figure shows the classes used in classic figure editor system[17]. FigureElement is an interface, and it can be either Point or Line. There is also a Display class that draws*

Figure on the screen. 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. Therefore, the display updating concern crosscuts four methods within two classes, and this concern has to be implemented in several classes. The code of example is shown in Figure 2.2. In this case, we add two methods update in both class Point and class Line. Assume that if there are several classes inherit from FigureElement, and we need to redraw the display after modify the value of visual properties in these classes. We have to add the new code in several classes. Fortunately Aspect-Oriented Programming fills this void.

2.1.2 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP)[18] is proposed for improving separation of concerns in software. Aspect-Oriented Programming allows the architect to address future potential requirements without breaking the core system architectures, and to spend less time on crosscutting concerns during the initial design phase, since they can be woven into the system as they are required without compromising the original design.

Aspect-oriented programming provides a solution to the crosscutting problem by supporting the modularization of crosscutting concerns in a novel construct called aspect. An aspect like a special class with functions that do not need to be directly referenced in another class in order to invoked. Programmer can localize the crosscutting concerns in the system from the core modules, class, and remove scattering and tangling problems from the program. AOP has following definitions: a join point is a point in the code or in the execution of a program at which concern crosscut the application. Generally, there are several join points for each concern. Join points can be considered as hooks within a program where the other parts of program can be conditionally attached and executed.

Join points are classified into two categories: dynamic join points and static join points. Dynamic join points are classified into call join points, execution join points, and field access join points[10]. Execution join points include method execution, initializer execution, constructor execution, static initializer execution, handler execution, and object initialization. Call join points include method call, constructor call, and object preinitialization. Field access join points include field references and field assignments. Static join points correspond with types to which new elements can be added.

A pointcut is that point of execution in an application where crosscutting concern needs to be applied. The expression of a pointcut is the pointcut descriptor. Pointcuts capture join points.

Advice is an additional piece of code that a programmer attempt to apply to an existing programming. The body of advice like a method, and contains the logic to be executed at each of the join points in a pointcut. Advice supports before, after, and around advice.

```
class Line implements FigureElement {
  private Point p1, p2;
  Point getP1() { return p1; }
  Point getP2() { return p2; }
  void setP1(Point p1) {
    this.p1 = p1;
  }
  void setP2(Point p2) {
    this.p2 = p2;
  }
  void draw(Display d){
    d.paintLine(getP1(), getP2());
  }
}
class Point implements FigureElement
{
  private int x = 0, y = 0;
  int getX() { return x; }
  int getY() { return y; }
  void setX(int x) {
    this.x = x;
  }
  void setY(int y) {
    this.y = y;
  }
  void draw(Display d){
    d.paintPoint(getX(), getY());
  }
}
```

```
aspect DisplayUpdating{

 pointcut change():
 execution
 (* FigureElement+.set*(..));

after() returning(): change() {
    Canvas.update();
 }

}
```

aspect

pointcut

advice

Figure 2.3: Implement of crosscutting concerns in figure editor system with AOP.

A before advice executed before a join point. An after advice executes after a join point. Around advice may replace the join point and execute instead of it. In addition there are two special situations, after returning and after throwing,

An aspect is a combination of a piece of code of pointcuts and advice. An aspect can also be more generally defined as a unit that encapsulates a crosscutting concern. The counterparts of aspects are components or base code, which are the functional units of code that only contain base actions, and aspect-oriented statements are not included. Components are units of code when written using functional, procedural or object-oriented languages.

EXAMPLE 2.2. *Figure 2.3 shows an example of implement of figure editor system by using AOP techniques. The updating display is implemented in an aspect. A pointcut descriptor in this aspect express the points in the execution flow after returning from methods setX, setY, setP1 or setP2, and the piece of advice associated with this pointcut then repaint the screen. By using aspect-oriented programming technique, we do not have to add new code in several classes.*

2.1.3 Aspect-Oriented Programming Languages

There are two types of AOP languages, one is programming language-specific AOP languages, the other is programming language-independent AOP languages[7]. And a lot

of these languages have been developed. For example AspectJ[17] and AspectC++[2] are the programming language-specific AOP languages which have been used widely. Weave.Net[11] is one of the language-independent AOP languages.

Weave.Net

Weave.Net is an independent Aspect-Oriented programming language. It avoids the tight coupling of aspects with components written in a particular programming language[11]. Weave.Net is implemented as a .Net component. The input of Weave.Net is a reference to a component assembly, the reference correspond to an XML document. The XML document contains specification for an aspect.

AspectC++

Aspect C++ is similar to AspectJ. Within AspectC++ designers are defined according to the syntax and semantics of c++. And in AspectC++ the wildcard "%" replaces wildcard " $*$ " which in AspectJ.

AspectJ

AspectJ is one of the prevalent and most mature aspect-oriented languages[17]. The AspectJ project started at the Xerox Palo Alto Research Center (PARC), AspectJ now is part of IBM's open source Eclipse project. AspectJ is an aspect-oriented programming extension of Java that supports crosscutting concerns. It 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. A summary of the syntax can be found in Appendix A. AspectJ uses a specific compiler which produces standard bytecode that can be executed on any Java virtual machine.

AspectJ mainly works on the interfaces of the classes. Join Points in AspectJ include method call, constructor call, method call execution, constructor call execution, field get, field set, exception handler execution, class initialization, and object initialization. There are several primitive pointcut designators within AspectJ. Pointcut descriptors are written as logical expressions defining which of these join points need to be picked out. The candidate is based on the name of the objects and methods involved and on their signature, using regular expressions. 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 the 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 pointcut can be combined with three logical operators.

A designator is used to expose the object to advice, or to narrow pointcut selection. Designators include this, target, and args. The designators of cflow and cflowbelow match join points within a given program flow, whereas, within and withincode match classes and methods. If and execution are dynamic designators. Advice in AspectJ supports before, after and around. Before advice can be utilized for checking preconditions and arguments. After advices can be classified into three different advices. The first one is unqualified after advice which runs no matter what the outcome of the join point. The second one is after returning advice which runs only if the join point returned normally. The third one is after throwing advice which executes if the join point ended by throwing an exception. Around advice runs instead of the join point and can invoke the join point using the special proceed syntax.

Aspects in AspectJ can be compared to classes in Java. It is the central unit of AspectJ, as of a class in Java. Pointcuts, advice, declarations and introductions are contained in an aspect. In addition to AspectJ elements, Aspects may contain fields, methods, and nested classes. In this study we implement our approach for AspectJ.

2.1.4 Fragile Pointcut Problem

The fragile pointcut problem[19] is similar to the fragile base class problem in object-oriented programming. Developers 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 determine 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. We briefly explain about two situations of the fragile pointcut problem, namely accidental join point captures and misses.

We use two versions of classic Figure Editor System as an example; one is the initial version of the system with no errors which is shown in Figure 2.4; and the other is the second version, in which pointcuts do not work due to program evolution. Figure 2.6 shows an example of second version system.

```
interface FigureElement{
  void draw(Display d);
  //...
}
class Point implements FigureElement{
  private int x =0, y=0;
  int getX(){ return x; }
  int getY(){ return y; }
  void setX(int xx){ this.x = xx; }
  void setY(int yy){ this.y = yy; }
  void draw(Display d){
    d.paintPoint(x, y);
  }
  //...
}
class Line implements FigureElement{
  private Point p1, p2;
  Point getP1() { return p1; }
  Point getP2(){  return p2; }
  void setP1(Point pp1){ this.p1 = pp1; }
  void setP2(Point pp2){ this.p2 = pp2; }
  void draw(Display d){
    d.paintLine(p1, p2);
  }
 //...
}
class Display{
  FigureElement f1, f2;
  void paintPoint(int x, int y){
    //...
  }
  void paintLine(Point p1, Point p2){
   //...
  }
  //...
}
```

Figure 2.4: An example of Figure Editor system(initial version)

In the initial version of the system, there are two different figure elements: a point element and a line element. Calls to `Point.setX` or `Point.setY` change the position of a point object. Similarly calls to `Line.setP1` or `Line.setP2` change the position of Line object. Methods `draw` in class `Point` and `Line` are used to implement the operations

```
1 aspect DisplayUpdating{
2   pointcut change(FigureElement felement):
3               execution (* FigureElement+.set* (..)) && target(felement);
4   after(FigureElement felement) returning(): change(felement) {
5     Canvas.update(felement);
6   }
7 }
```

Figure 2.5: `DisplayUpdating` Aspect

to paint itself, respectively. The display updating concern is implemented in the advice, shown in Figure 2.5. The pointcut `change` specifies executions of the methods which names starting with text "set" throughout the `FigureElement` class hierarchy. The advice in lines 4-6 redraws the screen whenever a figure object changes its visual properties.

Suppose that the program is changed to add two more operations in class `Point` in the second version: one is adding a `date` field to the class `Point`, the new field represents the last time the figure was saved on persistent store. And adding an associated `setDate` method to set its value. The other modification is adding a new method `moveBy`, this method represents move the position of a point from one location to another by add the new value of dx and dy. Figure 2.6 shows the second version of the class `Point`.

These changes make the advice in Figure 2.5 wrong in two situations. First is shown in Figure 2.7 that when we add a new method `setDate` into class `Point`, pointcut `change` will capture the new method `setDate` because its name starting with `set`, when after execute the method `setDate`, system would repaint the screen, however method `setDate` does not change any figure element's visual properties. Second is shown in Figure 2.8 that when we add a new method `moveBy` into class `Point`, pointcut `change` will not capture new method `moveBy`. Therefore if the Point is moved from one position to other, the screen will not be redrew because its name do not start with `set`.

The problem occur because the pointcut `change` does not specify correct join points; it accidentally specifies the execution of the new method `setDate` in the first case, and misses execution of the 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 our 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. Several approaches[5, 6, 8, 9, 13, 15, 16, 22, 23, 24, 25] have been proposed to attack the fragile pointcut problem will be discussed in next section.

2.2 Analysis-based Pointcut and Related Works

This section aims to describe a new pointcut construct language called analysis-based pointcut[6] and analysis of some relevant solutions of fragile pointcut problem. Finally, by comparing several related works, we indicate a gap between traditional name-based

```
class Point implements FigureElement{
  private int x =0, y=0;
  private sting date;
  int getX(){ return x; }
  int getY(){ return y; }
  void setX(int xx){ this.x = xx; }
  void setY(int yy){ this.y = yy; }
  void draw(Display d){
    d.paintPoint(x, y);
  }
  void setDate(string nowdate){
    this.date = nowdate;
  }
  void moveBy(int dx, ind dy)
   setX(x + dx);
   setY(y + dy);
  }
}
```

Figure 2.6: An evolved Figure Editor System(second version)

pointcut and new pointcut languages in current aspect-oriented software development. The necessity and significance of filling the gap will be discussed in this section, and indicate our study is the first one that attempt to bridge the gap.

The concept of fragile pointcut problem was proposed by Koppen et al.[19], pointcut is fragile due to the high coupling between aspect and base code's structure. AspectJ invented abstract aspects in order to reduce coupling. Aspects can contain abstract pointcuts which are defined by inheriting aspects. Therefore, abstract aspect can encapsulate all the advice code and reuse it. However, pointcuts in the concrete aspect still are fragile, even though abstract aspect can reduce coupling.

2.2.1 Analysis-based Pointcut

Analysis-based pontcut[6] is one of the approaches to overcome the fragility. We will utilize some examples to explain analysis-based pointcuts, and how they contribute to more robust than name-based one. Regular expression matching can be considered as a simplest analysis-based pointcut. Pointcus capture join points using regular expression when matching class/method/field names in the join points. This can be considered as an extension to wildcard-based type patterns in AspectJ. The following code of pointcut which proposed by [6] would match any method execution whose name consists of lowercase characters only.

```
pointcut executeLowercaseMethod():
```

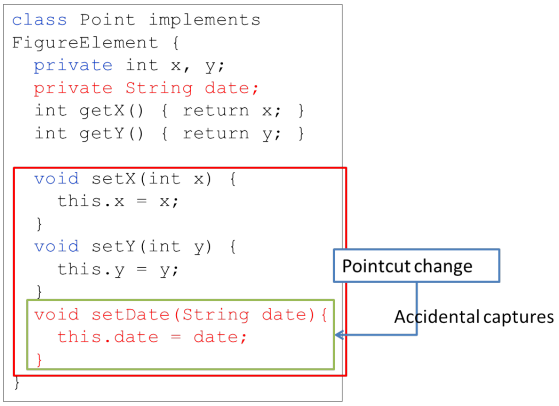

Figure 2.7: Accidental capture problem

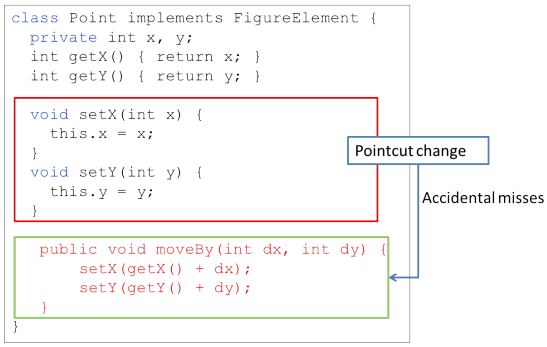

Figure 2.8: Accidental misses problem

Figure 2.9: Fragile Pointcut Problem.

```
  execution(.* .*\.[a-z]+(.*));
```

An analysis-based pointcut which utilized in our approach is defined by using static program analysis, and matched join points that satisfy the match pattern which the program analysis checks. Ignoring runtime overhead, we can achieve analysis-based pointcuts in existing AOP languages by using the conditional pointcut and introspective reflection as follows:

```
aspect DisplayUpdating{
  pointcut figureChange(FigureElement fe)
    :(execution (* *.*(..))
    && if(mayChange(thisJoinPoint)))
    && target(felement);
 after(FigureElement fe) returning(): figureChange(fe){
    Canvas.update(fe);
 }
}
```

The pointcut matches the method execution join points when a figure object changes its visual properties. The method `mayChange` takes a join point and return "true" if the method changes the visual properties of a figure object.

Compared to an aspect that enumerates all relevant method names or utilize wildcards instead of method names, the use of conditional pointcut clarifies the intention of the programmer. Moreover the aspect is robust against additions of new figure element classes and additions of new methods to existing figure element classes.

2.2.2 Related Works

Expressive pointcut languages

There are several relevant approaches which supporting analysis-based pointcut or similar to analysis-based pointcut that attempt to overcome fragile pointcut problem.

The Josh[9] and Alpha[21] are new AOP languages. Josh is a language like AspectJ. Josh allows the users to implement a new pointcut designator in Java, and it has its own weaver. However Josh does not support declarative pointcut specifications. Figure 2.10 shows an example of a user-defined designator updater which designed by [9].

```
1  //updater designator:
2  updater("FigureElement", "redraw");
3
4  //static method which implement the updater designator:
5  static boolean updater(MethodCall mc, String[] args, JoshContext jc){
6    CtClass root = jc.getCtClass(args[0]);
7    String mname = args[1];
8    CtMethod mth = mc.getMethod();
9    //skip if the method is redraw().
10   if(mth.getName().equals(mname))
11      return false;
12   Hashtable fields = enumerateFields(jc, root, mname);
13   update = false;
14   mth.instrument(new ExprEditor(){
15     public void edit(FieldAccess expr){ ... }
16   }):
17   return update;
18 }
```

Figure 2.10: An example of Josh

Method updater returns false if the called method mth is redraw. Otherwise, it invokes enumerateField for enumerating the fields that the redraw methods in a subclass of FigureElement read.

Alpha provides rich program information to user defined pointcuts. The Alpha aspect language use a logic programming language for the specification of pointcuts. Figure 2.11 shows an example of enum pointcut in Alpha.

```
1 class DisplayUpdate {
2   Display d;
3   after set(P, x, _ ); set(P, y, _); set (P, 'p1', _); set(P, 'p2', _),
4            instanceof(P, 'FigureElement') {this.d.draw(P);}
5   //...
6 }
```

Figure 2.11: An example of pointcut in Alpha

The enum piontcut (line 3) enumerates all assignments to fields that potentially affect drawing behavior, the fields namely `x`, `y`, `p1`, `p2` are in any object `P` of type FigureElement. This pointcut utilize the names of fields to identify the relevant assignments. Nevertheless, this new pointcut language is difficult to be written by programmer who do not familiar with logic programming language, and its dynamic execution model needs a complex compilation framework to achieve efficient performance.

Although some expressive pointcut languages make pointcut definition much less fragile, they do not solve the problem completely. A pointcut definition still need to refer to specific base program structure or behavior to match its join points. In order to deal with the fragile based on structural dependencies, Kellens et al.[15] propose a novel pointcut construct language so-called model-based pointcuts. These new pointcuts are decoupled from the base program structure, for conceptual model instead. For example, assuming that the conceptual model contains a classification of all `figurechange` methods in the implementation of the figure editor system. The model-based pointcut that captures all join points to `figurechange` methods could be defined as:

```
1 pointcut figurechange():
2    classifiedAs(?methSignature, FigurechangeMethods) &&
3    call(?methSignature);
```

where the expression `classifiedAs(?methSignature, FigurechangeMethods)` matches all methods that are classified as `figurechange` methods in the conceptual model of the figure editor system, and the variable `?methSignature` is bound to the method signature of such a method. This Pointcut definition explicitly refers to the concept of a `figurechange` method rather than attempting to capture concept by depending on implicit rules about the base system's implementation structure. Therefore, this pointcut does not need to be changed when the base system is evolved. Obviously, this new pointcut language is more powerful, however, similar to the Alpha, this pointcut language is also difficult to be written by programmers who do not familiar with new language.

Annotation

An alternative solution that has been proposed is to define pointcuts in terms of explicit annotations in the code. Silva et al.[23] propose a solution relies on non-invasive and non-scattered annotations to solve the fragile pointcut problem. The central components of the proposed solution are annotator aspects that superimpose annotations to the based

code in a non-invasive way. Figure 2.12 shows an example of annotation-aware interface and annotator aspects in this approach.

```
1 //annotation-aware interface associated to the class Point
2 class Point{
3   DisplayStateChagne void setX(int);DisplayStateChange void setY(int);
4 }

6 //annotator aspect
7 aspect DisplayStateChange{
8   declare method :public void Point.setX(int) :  DisplayStateChange;
9   declare method :public void Point.setY(int) :  DisplayStateChange;
10  //...
11 }
```

Figure 2.12: An example of annoatation-aware interface and annotator aspects

Line 1-4 implements annotation-aware interface and line 6-11 implements annotator aspect. By using this approach, the following pointcut is decoupled from the base system, but it also requires programmers to annotate correctly all methods that change the state of the display.

```
1 pointcut change() ;
2     execution(void DisplayStateChange *.*(..))
```

Moreover, this solution is particularly recommended in two situations: when it is possible to correlate annotations, and it is possible to restrict the scope of an annotation.

Integrated Tools and Frameworks

Pointcut rejuvenation[16] and Pointcut-Doctor[25] are integrated tools that assist programmer to write correct pointcuts. Pointcut rejuvenation picks up a lot of suggested join points which are ranked by the value of confidence after program evolved. Pointcut-Doctor is an extension of AJDT tools that helps programmers write correct pointcuts by providing immediate diagnostic feedback. This approach present an algorithms to compute two kinds of information about AspectJ pointcuts: almost matched join point shadows and explanations. The algorithm has two variants, and their algorithm captures join points by changing the names in the pointcut slightly.

Anbalagan et al.[5] propose a framework to generate pointcut mutants with different strengths, and assist developers inspect the pointcut and their join points conveniently. For example, consider the arguments "(int, float, String)". The mutants formed for this argument will be (int, float, ..), (int, .., String), (.., float, String), (.., float, ..), (int, ..), (.., String), and (..).

However these solutions cannot update original pointcut, and need programmers to discriminate correct join points, then revise original name-based pointcuts manually.

Crosscutting interface (XPIs)

XPIs are explicit, abstract interface that decouple aspects from details of advised code[12]. XPIs define contracts that programmers must observe. On the other hand, aspect developers must rely on the syntactic part of XPIs to implement advices that do not directly reference source code elements. Pointcuts based on XPIs are also more robust, since the base code have to adhere to the defined design rules even after changes. Design rules enforced by XPIs are implemented in AspectJ.

2.2.3 Summary

Solution	Type	Objective	Problem
Josh[9]	Expressive pointcut language	Get robust join points after program evolved	Difficult to write
Alpha[21]	Expressive pointcut language	Get robust join points after program evolved	Difficult to write
Pointcut Mutants[5]	Framework	Assist programmer inspect the pointcut and their join points conveniently	Need to modify original pointcut manually
Annotation Pointcut Language[23]	Annotation	Get robust join points after program evolved	Need Programmer to decide whether should be annotated
Analysis-based Pointcut[6]	Expressive pointcut language	Get robust join points after program evolved	Difficult to write
Model-based Pointcut[15]	Expressive pointcut language	Get robust join points after program evolved	Difficult to write
Pointcut Rejuvenation[16]	Integrated tool	Assist programmer write correct pointcuts after program evolved	Need to modify original pointcut manually
Pointcut Doctor[25]	Integrated tool	Assist programmer write correct pointcuts after program evolved	Need to modify original pointcut manually

Table 2.1: Comparison of Related Works

To summarize we compare several different solutions on overcome pointcut fragile problem, as shown in Table 2.1. We find that new pointcut construct language which improve

their expressiveness are very different from the original pointcut language, thus, the new robust pointcut languages are difficult to be written by programmers who do not familiar with new pointcut languages. In addition integrated tools and framework can only assist programmer to check the correctness of join points in order to revise original pointcuts manually. In consequently, there is a gap between traditional name-based pointcuts and other new pointcut languages, and no approach can update original pointcuts automatically after program evolves. We propose a framework called Nataly can translate name-based pointcut into analysis-based pointcut automatically in order to bridge the gap.

Chapter 3

Proposed Framework-Nataly

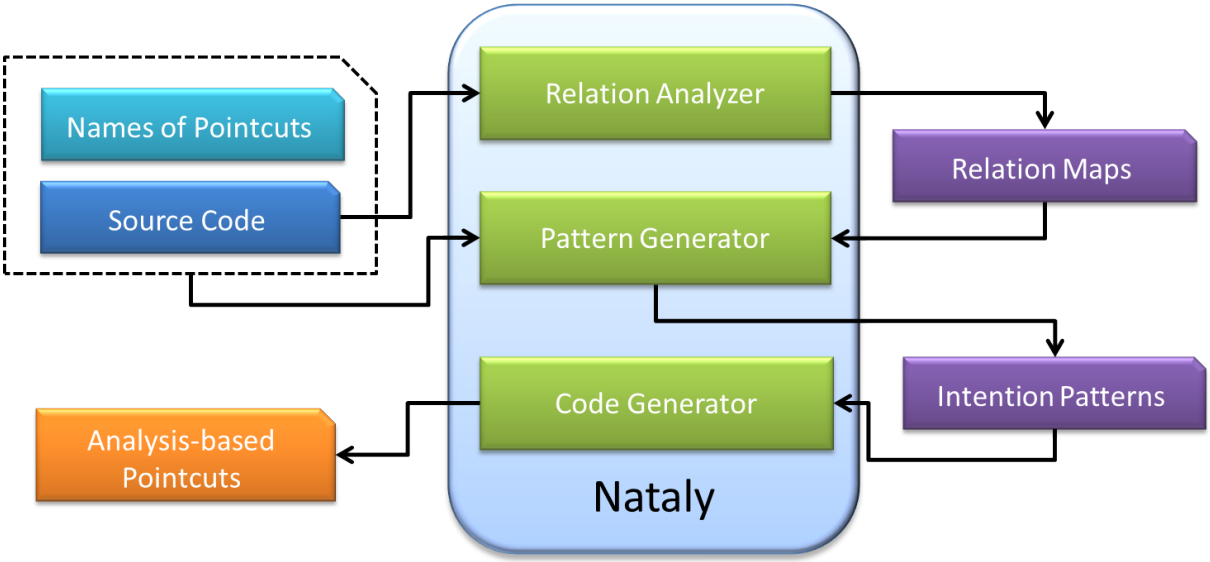

Figure 3.1: Overview of our framework

3.1 Overview of Proposed Framework

This section explains Nataly, a framework for translating name-based pointcuts into analysis-based pointcuts automatically.

Figure 3.1 shows the overview of the framework. It consists of three significant components, namely relation analyzer, pattern generator and code generator. The framework takes classes, aspects and the names of pointcuts as input. The names are used to identify which pointcuts are to be translated into analysis-based pointcuts. Then the relation analyzer generates eleven relation maps that represent relationships among fields, methods, classes and interfaces. These maps are used by the pattern generator to extract the

intention patterns for each pointcut to be translated. And finally, the code generator generates analysis-based pointcuts by using the information.

3.2 Assumptions

Before introduce the core of our framework, we will indicate some significant assumptions first. Because our framework presented in this work is under these key assumptions.

- We assume that the initial pointcut which to be transferred is specified correctly. Because the component of pattern generator needs the initial pointcut as an input, and extract intention pattern by using relationship trees which are built correspond to a given pointcut. If the pointcut is not correct, then its counterpart analysis-based pointcut will not capture the correct set of join points. Specifically, we assume that advice is created to materialize the implementation of concern which crosscuts the underlying base code. So the pointcut bound to the advice should quantify over the join points which correspond to this crosscutting concern.

- We assume that aspects are indeed separate from the base code, and advice may only apply to join points associated with classes, interfaces, and other Java types.

- Furthermore, we assume that the original source code successfully compiles under an AspectJ Development Tools (AJDT) compiler.

Lastly, we assume that the pointcut which need to be transferred is not an anonymous one. Because our framework need the name of pointcut and then generates its counterpart, if the pointcut does not have a name, it is difficult to know which pointcut need to be transferred. In the future, we plan to also support pointcut designators as an input in order to support anonymous pointcut transformation.

3.3 Relation Analyzer

The relation analyzer generates eleven relation maps by analyzing the source code. The relation maps describe the relationship between two program elements. These maps are defined in Table 3.1. *type* ranges over the names of classes and interfaces, and *name* ranges over the names of methods and fields. Notice that there are two types of relation maps. One is adjacent relation map which means the relationship between element A and B follow the forward direction. For instance, relation map of mcall between A and B represents that method A calls method B; On the other hand, its counterpart called opposite relation map. The opposite relation map often describes the same relationship between A and B, but in opposite direction. For instance, relation map of mcallee between A and B represents that method A is called by method B. If we analyze adjacent relation map, we call this analysis as a forward analysis. Otherwise, if we analyze opposite relation map, we call this analysis as a backward analysis. The details of each relation map are described in Table 3.2.

map name	key	value
fdecl	*type*	*list of name*
mdecl	*type*	*list of* (*type*, *name*, *list of type*)
fset	(*type*, *type*, *name*, *list of type*)	*list of* (*type*, *name*)
fget	(*type*, *type*, *name*, *list of type*)	*list of* (*type*, *name*)
mset	(*type*, *name*)	*list of* (*type*, *type*, *name*, *list of type*)
mget	(*type*, *name*)	*list of* (*type*, *type*, *name*, *list of type*)
mcall	(*type*, *type*, *name*, *list of type*)	*list of* (*type*, *name*, *list of type*)
mcallee	(*type*, *type*, *name*, *list of type*)	*list of* (*type*, *name*, *list of type*)
mconcretize	(*type*, *type*, *name*, *list of type*)	*list of* (*type*, *name*, *list of type*)
tconcretize	*type*	*list of type*

Table 3.1: Overview eleven relation maps

map name	description
fdecl	represents a set of fields declared by a class.
	Its key is a class name, and the value is a list of field names.
mdecl	represents a set of methods declared by either a class or an interface.
	Its key is a class name, and the value is a list of triples (type, name, list of type) that identifies a method.
cdecl	represents a set of classes declared by a class.
	Its key is a class name, and the value is a list of class names.
fset	represents a set fields that are to be updated within a method.
	Its key, (type, type, name, list of type), identifies a method; type1 is the return type; type2 is the class name that declares the method; name is the method name; and list of types is the argument tpes. The value list of (type, name) is the set of undered fields, where type is a class that declares the field and name is its name.
fget	represents a set of fields whose values gotten within a method.
	Its key is a method, and its value is a set of accessed fields.
mset	represents a set of methods that updates the value of field.
	Its key is an updated filed, and its value identifies a set of methods.
mget	represents a set of methods that gets the value of a field.
	Its key is an accessed field, and its value identifies a set of methods.
mcall	represents a set of methods called within a method.
	Its key identifies a method, and its value identifies a set of called methods.
mcallee	represents a set of methods that calls a method
	Its key identifies a called method, and its value identifies a set of methods.
mconcretize	represents a method that overrides/implements a list of method.
	Its key identifies a method, and its value identifies a set of methods.
tconcretize	represents a class extended by a class, or set of interface extended by an interface/implement by a class.
	Its key is a class/Interface name, and the value is a list of class and interface names.

Table 3.2: Details of relation maps

The *fdecl*, *mdecl*, *cdecl*, *fset*, *fget* and *mcall* are adjacent relation maps. The rest relation maps of *mcallee*, *mset*, *mget*, *mconcretize* and *tconcretize* are opposite relation maps.

Analysis	ralation map	map name	key	value
Forward Analysis	`Point` $\xrightarrow{mdecl}$ `moveBy`	*mdecl*	`Point`	`moveBy`
	`moveBy` $\xrightarrow{mcall}$ `setX`	*mcall*	`moveBy`	`setX`
	`setX` $\xrightarrow{fset}$ `x`	*fset*	`setX`	`x`
	`moveBy` $\xrightarrow{mcall}$ `setY`	*mcall*	`moveBy`	`setY`
	`setY` $\xrightarrow{fset}$ `x`	*fset*	`setY`	`y`
Backward Analysis	`x` $\xrightarrow{mget}$ `Point.draw`	*mget*	`x`	`Point.draw`
	`Point.draw` $\xrightarrow{mconcretize}$ `FigureElement.draw`	*mconcretize*	`Point.draw`	`Figure-Element.draw`
	`y` $\xrightarrow{mget}$ `Point.draw`	*mget*	`y`	`Point.draw`

Table 3.3: A part of relation maps in figure editor system(1)

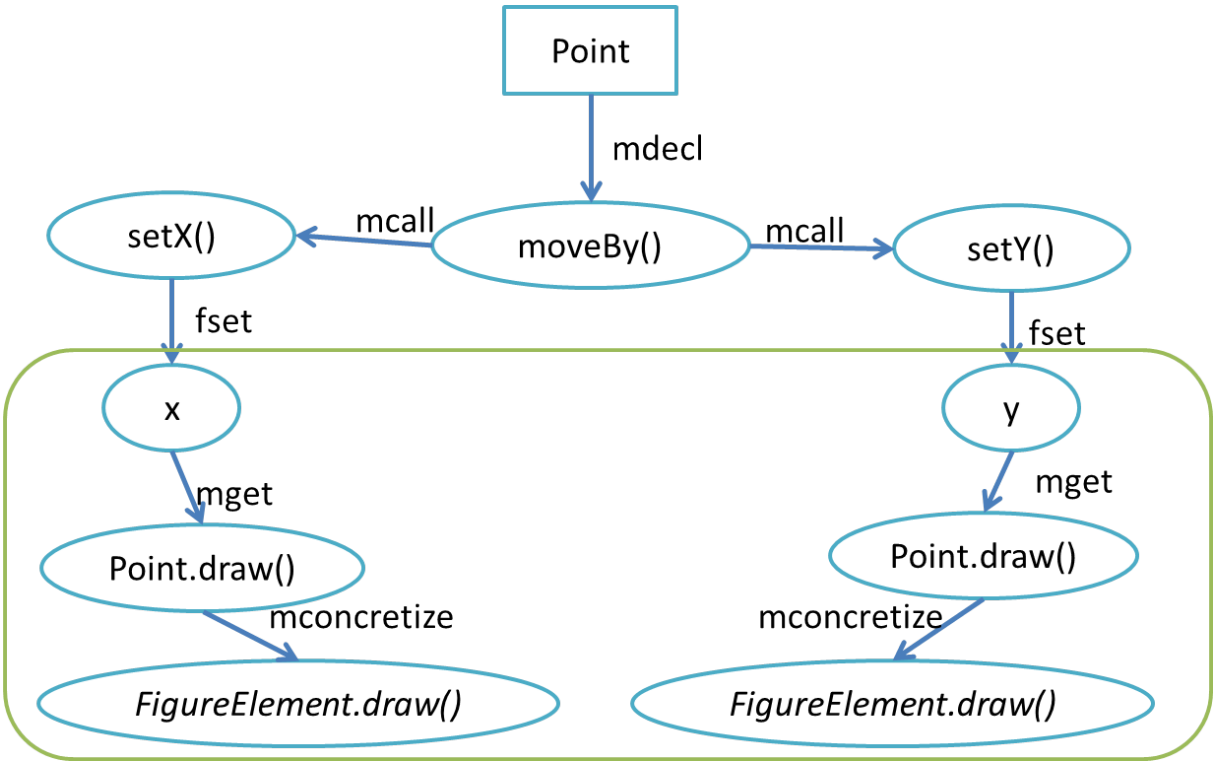

Figure 3.2: A part of relation maps in figure editor system(2)

EXAMPLE 3.1. *Table 3.3 and Figure 3.2 show a part of relation maps within figure editor system in different views. By using forward analysis, we can find that class* `Point` *declares method* `moveBy`*, and method* `moveBy` *calls method* `setX`*, and method* `setX` *sets the value of field* `x`*. Method* `moveBy` *also calls another method* `setY`*, and method* `setY` *sets the value of field* `y`*. On the other hand, by using backward analysis, we can see that the value of filed* `x` *is gotten by method* `Point.draw`*, and the method* `Point.draw` *implements the method* `FigureElement.draw` *in an interface. The value of field* `y` *is gotten by method* `Point.draw`*, and the method* `Point.draw` *implements the method* `FigureElement.draw` *in an interface. In addition relation maps will be persisted as an Extensible Markup Language (XML). Figure 3.3 shows the example of a mcall relation map between method* `moveBy` *and method* `setX` *in an XML document.*

```
<?xml  version="1.0" encoding="UTF-8"?>
<java  version="1.6.0_20" class="java.beans.XMLDecoder">
  <object class="jaist.info.aspectj.analyisispc.metadata.SourceShadow">
    <void property="projectName">
      <string>/home/wanglin/runtime-analysispcRun/FigureEditor/src/figure/
      DisplayUpdating.aj</string>
    </void>
    <void property="shadowList">
      <void method="put">
        <string>figure.Point.moveBy()</string>
        <object id="SourceShadowNode1" class="jaist.info.aspectj.analysispc.
        metadata.SourceShadowNode">
          <void property="mcallList">
            <void method="put">
              <string>figure.Point.setX()</string>
              <object idref="SourceShadowNode2"/>
            </void>
           <!-- ...... -->
          </void>
        </object>
      </void>
    <!-- ...... -->
    </void>
  </object>
</java>
```

Figure 3.3: An example of XML document

3.4 Pattern Generator

This section aims to explain the component of pattern generator. The pattern generator generates intention patterns using the relation maps created by the relation analyzer. An intention pattern is relevant to a given pointcut, and intuitively represents the relationships among elements associated with matched join point shadows. And the primary pattern is first proposed by point rejuvenation[16]. We utilize these patterns to express the essence of the programmer's intentions with regard to the input pointcut. There are three phases in pattern generator: build relationship trees, extract intention pattern and persist pattern format to Tregex pattern format[3].

3.4.1 Build Relationship Trees

In this section we present how to build the relationship trees. First, we obtain join point shadows correspond to a given pointcut by the support of AJDT compiler (`http://eclipse.org/ajdt`). Second, we utilize the join point shadows to find out the associated java elements from the relation maps. Third, we utilize java elements which associated with join point shadows as a root to build the tree by using relation maps. When build the trees there are four principles of rules need to be complied with:

- When we forward build the tree, if the last vertex is a method, then we stop to build this branch, and use current method as a target in this branch.
- When we forward build the tree, if the last vertex is a field, we save this vertex as a temporary target, and then continue to backward build the tree by using opposite relation maps.
- When we backward build the tree, if the current vertex is an abstract method within an abstract class or an interface, then we stop to build this branch, and use current abstract method as target in this branch.
- When we backward build the tree, if we cannot reach an abstract method in current branch, then we stop to build this branch, and use the temporary target as a genuine target in this branch.

The vertex on the tree is represented by V_{pc} or V_{npc}; the edge of the tree is represented by R. The tree likes a log that records the behavior of the V_{pc}. V_{pc} is an element associated with join point shadows in the program matched by a given pointcut; V_{pc} will become join point shadows ultimately. V_{npc} is an element in the program which isn't matched by the given pointcut. Similarly, R represents a relationship between two elements in the program. The last R along with target represent the objective of the behavior of the V_{pc}.

EXAMPLE 3.2. *Figure 3.4 illustrate how to build a relationship tree from figure editor system. In order to build the tree we need to utilize relation maps which generated by relation generator, and the example of relation maps have been described in Figure 3.3. In figure editor system, we have a method execution pointcut called* `change`*, First, we*

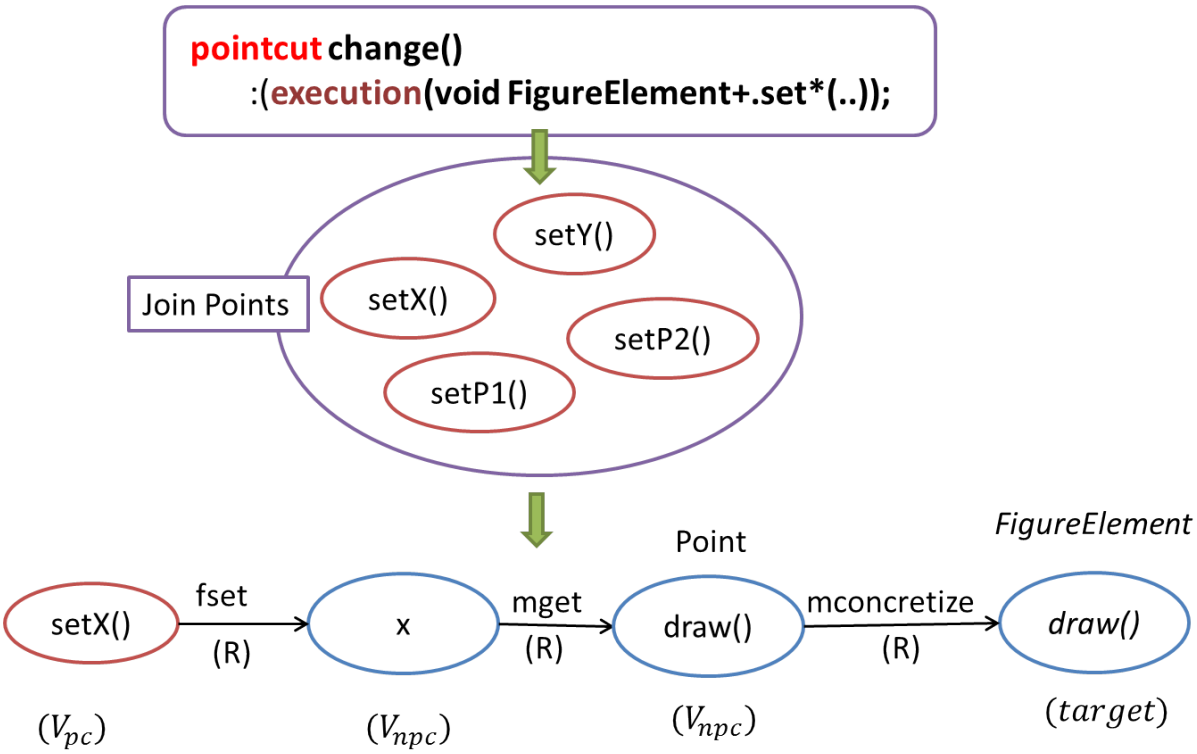

Figure 3.4: An example of relationship tree from figure editor system

capture four join point shadows, method `setX`*, method* `setY`*, method* `setP1` *and method* `setP2` *which matched by the pointcut* `change` *by the support of AJDT compiler. We use method* `setX` *as a root to illustrate how to build the tree in this example. By complied with the principles of rules which mentioned before, from method* `setX` *we find that there is a relationship called fset between method* `setX` *and field* `x` *in class* `Point`*. Thus, we add the vertex* `x` *into the tree. Next, we find that vertex* `x` *is a field and it is the last vertex in this tree, therefore follow the second principle, we save vertex* `x` *as a temporary target, and continue to backward build the tree by using opposite relation maps. Next we find that there is a relationship called mget between field* `x` *and method* `Point.draw`*, and method* `Point.draw` *maps to the method* `FigureElement.draw` *called mconcretize. thus,* `FigureElement.draw` *is an abstract method, according to the third principle, we stop build the tree, and set the vertex* `FigureElement.draw` *as a target. Now, the tree has been built completely.* `setX` *corresponds to* V_{pc}*,* `x` *and* `Point.draw` *correspond to* V_{npc}*, and labels fset, mget and mconcretize denote relationship R.*

Figure 3.5 shows two different build ways in this example. As we have mentioned before, we have two types of relation maps, namely adjacent relation map and opposite relation map, respectively. If we use adjacent relation maps to build the tree, then we call this way as forward build. On the other hand, if we use opposite relation maps to build the tree, then we call this way as backward build. In the Figure 3.5, we build the tree from vertex `setX` *to vertex* `x` *by the forward way, from vertex* `x` *to vertex* `FigureElement.draw`*, the tree is built by the backward way.*

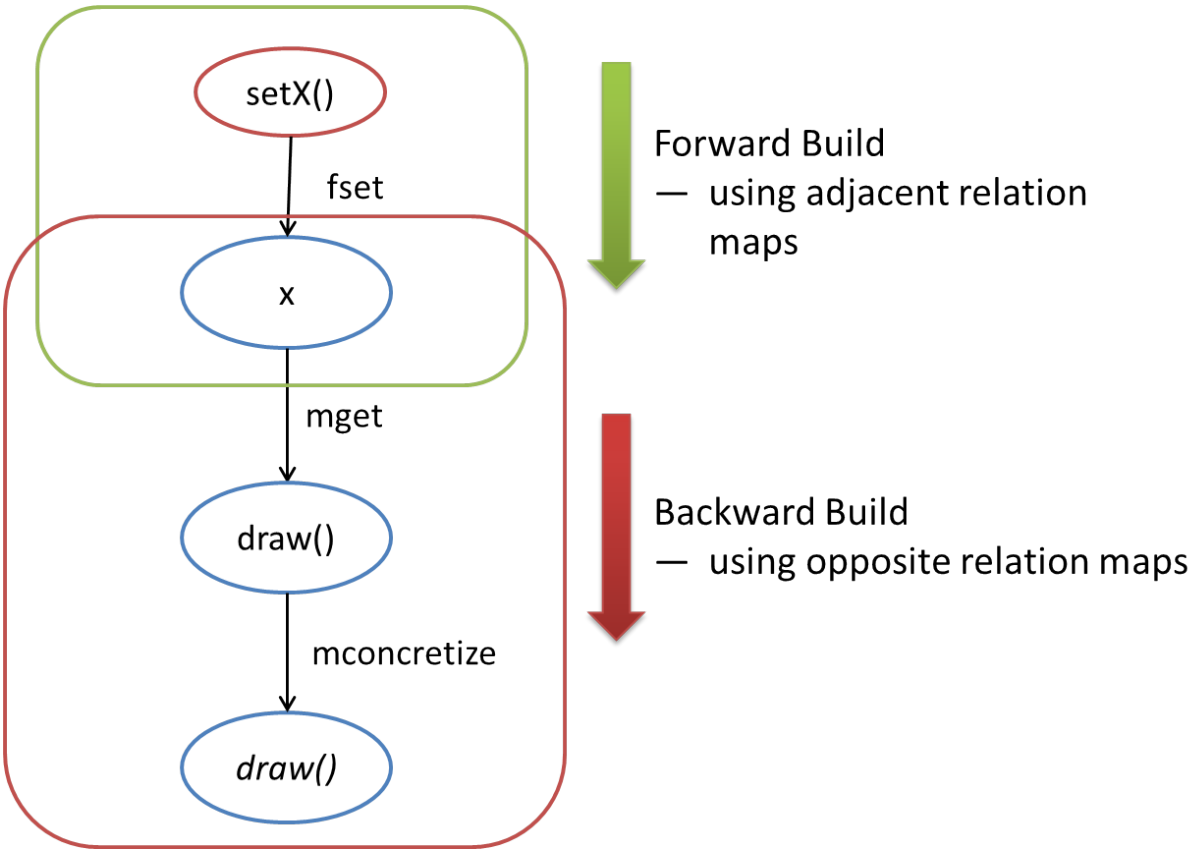

Figure 3.5: An example of Forward Buid and Backward Build

3.4.2 Extract Intention Pattern

Intention Pattern

An intention pattern is relevant to a given pointcut, and intuitively represents the relationships among elements associated with matched shadows. The purpose of these patterns is to approximate the essence of the programmer's intentions behind the original pointcut. As we have mentioned before, the relationship tree represents the behavior of the associated join point shadows matched by a given pointcut. The intention pattern is generated from several relationship trees through analyze their properties, including keeping common properties, disjunctioning different objective and generalizing insignificant vertex (V_{npc}).

In addition to four elements in relationship tree, there are three other elements in our intention pattern, namely $Wildcard_{pc}$, $Wildcard_{npc}$ and $Wildcard_R$. Wildcards are used to match patterns with other trees in order to obtain possible join point shadows.

Pattern Extraction

We have introduced in section 3.4.1, there are four elements in our relationship trees, namely V_{pc}, V_{npc}, R and target. During pattern extraction phase, V_{pc} will be replaced

by $Wildcard_{pc}$, and V_{npc} may be replaced by $Wildcard_{npc}$ within an intention pattern. Similarly, R may be replaced by $Wildcard_R$. For example, ?* $\xrightarrow{R1}$? $\xrightarrow{R2}$? $\xrightarrow{R}$ *target* is a relationship tree, where notation ?* denotes $Wildcard_{pc}$, notation ? denotes $Wildcard_{npc}$.

Algorithm 1 defines a function that can extract pattern from the relationship trees. Initializes a set P to be an empty set of patterns to be returned at line 1. The algorithm then separate the relationship trees into several groups according to the same target by function ClassifyTree. Π is a list that contains several sets of trees, each set contains a group of trees that have the same target. Notice that each tree may be copy to many groups due to the multi-target. For example, if a tree has three branches, and each branches has three different targets. And there are three groups correspond to each target. Then the tree will be copy to three groups. Function CutUnusefulBranch is used to remove branches which target do not exist in. After cut the branches by function CutUnusefulBranch, each tree only has one branch that contains the vertex of target.

In other words, after cut the unuseful branches, π' represents a set of path that has the same target. $ExtractVPattern$ is used to abstract vertexes from the tree. Algorithm 2 defines ExtractVPattern function. It receives one parameter, namely π' which represents a set of paths which has the same target. To do so, we first initialize a set P and a pattern α to be an empty value. Next, algorithm checks each vertex in the tree, if the vertex is V_{pc}, then algorithm adds the vertex and its followed relation R into the pattern α. And then continue to check the vertex in sequence. If all the trees do not contains this vertex, algorithm uses $Wildcard_{npc}$ replace the vertex of V_{npc}, and adds a pair of new vertex and its followed relation R into the pattern α. Otherwise, algorithm keeps this vertex and adds it and its followed relation R into the pattern α. After abstract every vertex, algorithm disjunctions of each pattern α to the set P. Finally, the function $ExtractVPattern$ retruns set P. On the other hand, function ExtractRPattern is used to abstract relationship from the tree. Algorithm 3 defines $ExtractRPattern$ function. It receives one parameter, namely π' which represents a pattern, in fact it is a set of paths that consist of $Wildcard_{pc}$, relationship R, $V_{pc}/Wildcard_{npc}$, and target. To do so, we initialize a set P and a pattern β to be an empty value. Next algorithm checks if relationship R satisfied Condition 1 or Condition 2, then it replaces relationship R by $Wildcard_R$. Otherwise, it keeps relationship R in the pattern.

CONDITION 1. *Adjacent Relationship Abstraction Conditions: to a given adjacent relationship r and its index of path i.*

- *All the vertexes within all the different trees must same in the index of i+1.*
- *At least one tree does not contain relationship r in the index of i.*

CONDITION 2. *Opposite Relationship Abstraction Conditions: to a given opposite relationship r and its index of path i.*

- *All the vertexes within all the different trees must same in the index of i.*
- *At least one tree does not contain relationship r in the index of i.*

Other process is similar to the function $ExtractVPattern$. Next we use function CombinePattern to combine the repeated patterns and remove redundant patterns. Finally, we generalize the pattern by using function $GeneralizePattern$. Function $GeneralizePattern$ receives one parameter, namely P, which represents a pattern. In order to generalize the pattern, algorithm adds a general element between two adjacent vertexes in pattern. A general element consists of arbitrarily number of $Wildcard_{npc}$ and $Wildcard_R$. In other words, the hierarchical relationship between two original vertexes is changed from parent to ancestor.

Algorithm 1 Extract Pattern

1: **function** ExtractPattern(T) //T is the set of relationship trees which generated in the first phase
2: $P \leftarrow \varnothing$ //the set of patterns to be returned, initially empty
3: $P' \leftarrow \varnothing$
4: $\pi' \leftarrow \varnothing$
5: $\Pi' \leftarrow \varnothing$
6: $\Pi \leftarrow ClassifyTree(T)$
7: //Π is a list of sets of trees which classified by their target
8: **for all** $\pi \in \Pi$ **do** //each π is a set of tree which has the same target
9: $\pi' \leftarrow CutUnusefulBranch(\pi)$
10: // π' is a set of paths which has the same target
11: $\Pi' \leftarrow ExtractVPattern(\pi')$ //Replace some vertexes by wildcard
12: $P' \leftarrow P' \cup \Pi'$
13: **end for**
14: **for all** $\pi' \in P'$ **do**
15: $\Pi' \leftarrow ExtractRPattern(\pi')$ //Replace relation R by $wildcard_R$
16: $P \leftarrow P \cup \Pi'$
17: **end for**
18: $P \leftarrow CombinePattern(P)$
19: $P \leftarrow GeneralizePattern(P)$
20: **return** P
21: **end function**

Algorithm 2 Extract Vertex Pattern

```
1: function ExtractVPattern(π′)
2:     P ← ∅
3:     α ← ∅
4:     for all t ∈ π′ do
5:         for all v ∈ t do
6:             r ← GetRByVertex(v, t)
7:             if IsVPC(v) then
8:                 α ← α + (?∗)^r
9:             else if IsTarget(v) then
10:                α ← α + (v)^r
11:            else if AllTContainsV(v, t) then
12:                α ← α + (v)^r
13:            else
14:                α ← α + (?)^r
15:            end if
16:        end for
17:        P ← P ∪ α
18:    end for
19:    return P
20: end function
```

Algorithm 3 Extract Relation Pattern

```
1: function ExtractRPattern(π')
2:     P ← ∅
3:     β ← ∅
4:     for all t ∈ π' do
5:         v ← GetVertexByIndex(i, t)
6:         β ← β + (v)
7:         n ← getMaxLength(t)
8:         for i ← 1, n − 1 do
9:             r ← GetRelation(i, t)
10:            v ← GetVertexByIndex(i + 1, t)
11:            if AllTContainsR(r, i, π') then
12:                β ← β + (v)^r
13:            else
14:                if IsAdjacentR(r) then
15:                    if IsAllVertexSame(v, i + 1, π') then
16:                        β ← β + (v)^?
17:                    else
18:                        β ← β + (v)^r
19:                    end if
20:                else
21:                    v' ← GetVertexByIndex(i, t)
22:                    if IsAllVertexSame(v', i, π') then
23:                        β ← β + (v)^?
24:                    else
25:                        β ← β + (v)^r
26:                    end if
27:                end if
28:            end if
29:        end for
30:        P ← P ∪ β
31:    end for
32:    return P
33: end function
```

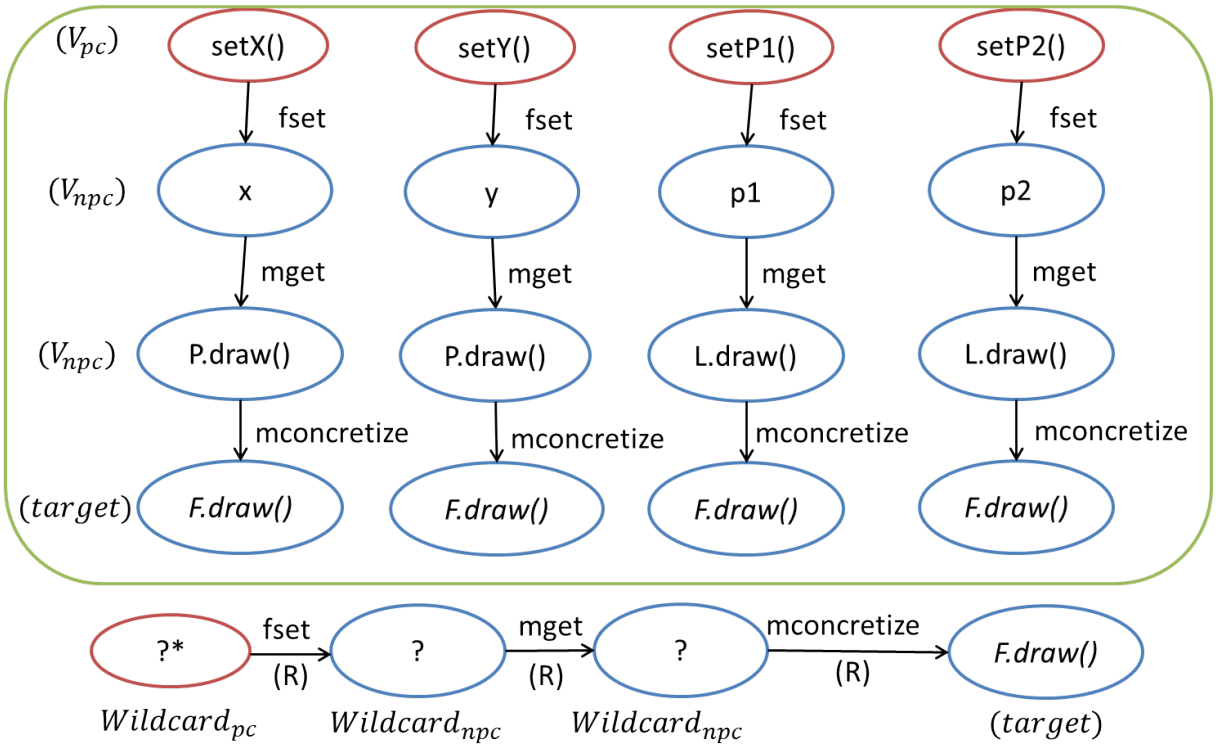

Figure 3.6: An example of extract interntion pattern

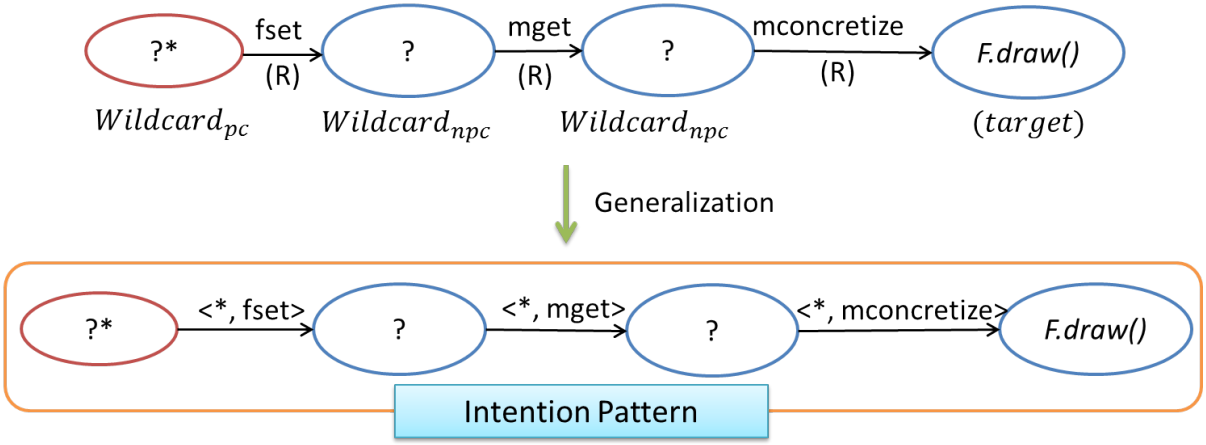

Figure 3.7: An example of generalize intention pattern

EXAMPLE 3.3. *Figure 3.6 illustrates how to extract pattern from the relationship trees*

in figure editor system. In this example, all the trees have a same target, so we only have one group of pattern. And every tree has only one branch, so we do not need to cut the unuseful branches. Next, we use $Wildcard_{pc}$ *replace vertex* `setX`, `setY`, `setP1` *and* `setP2` *in four trees. After that, we get four temporary patterns:* $?* \xrightarrow{fset} x \xrightarrow{mget} P.draw \xrightarrow{mconcretiize} F.draw$; $?* \xrightarrow{fset} y \xrightarrow{mget} P.draw \xrightarrow{mconcretiize} F.draw$; $?* \xrightarrow{fset} p1 \xrightarrow{mget} L.draw \xrightarrow{mconcretiize} F.draw$; $?* \xrightarrow{fset} p2 \xrightarrow{mget} L.draw \xrightarrow{mconcretiize} F.draw$. *Next, the other vertex will be abstracted by* $Wildcard_{npc}$. *Because vertex* `x` *is not exist in other trees, vertex* `x` *is replaced by* $Wildcard_{npc}$. *Similarly vertex* `y`, `p1` *and* `p2` *are also not exist in other trees, and they are also replaced by* $Wildcard_{npc}$. *For the same reason, vertexes of P.draw and L.draw are replaced by* $Wildcard_{npc}$. *Now, we get four temporary patterns:* $?* \xrightarrow{fset} ? \xrightarrow{mget} ? \xrightarrow{mconcretiize} F.draw$; $?* \xrightarrow{fset} ? \xrightarrow{mget} ? \xrightarrow{mconcretiize} F.draw$; $?* \xrightarrow{fset} ? \xrightarrow{mget} ? \xrightarrow{mconcretiize} F.draw$; $?* \xrightarrow{fset} ? \xrightarrow{mget} ? \xrightarrow{mconcretiize} F.draw$. *We find that these four patterns are identical with each other. In addition, notice that the relationship R does not satisfies our conditions, because of the identical name. Therefore, we keep original relationship R in the pattern. Finally, by combined the repeated pattern and removed redundant pattern we get the final pattern:* $?* \xrightarrow{fset} ? \xrightarrow{mget} ? \xrightarrow{mconcretiize} F.draw$.

Figure 3.7 explains the generalization of pattern in figure editor system. In this example, between first vertex and second vertex, we add a general element, namely, $< *, fset >$, *Notation* $< *, fset >$ *represents that there are arbitrarily number of* $Wildcard_{npc}$ *and* $Wildcard_R$ *between these two vertexes, and last relationship is restricted to fset. For example, it is equivalent to:* $?* \xrightarrow{?} ? \xrightarrow{?} ? \cdots \xrightarrow{fset} ? \xrightarrow{mget} ? \xrightarrow{mconcretize} F.draw$. *Similary, generalize other parts of the pattern, Finally, we obtain the intent pattern:* $?* \xrightarrow{<*,fset>} ? \xrightarrow{<*,mget>} ? \xrightarrow{<*,mconcretiize>} F.draw$.

3.4.3 Tregex Pattern Format

We integrate Tregex (`http://nlp.stanford.edu/software/tregex.shtml`) which supported by Stanford Natural Language Processing Group as a tree match engine to help us match the trees by using the intention pattern. Tregex is a utility for matching patterns in trees, based on tree relationships and regular expression matches on nodes. Tregex needs a specific format of the pattern. So we need to persist our intention pattern to Tregex Pattern format.

The basic units of Tregex are Node Descriptions. Descriptions match node labels of a tree, such as a Literal string "NP", Symbol "|" separates disjunction of literal strings, like NP|PP|VP. Tregrex also support regular expression. Notice that wildcard symbol "__" (two underscores) are used to match any node. Moreover, descriptions can be negated with "!". Figure 3.8 shows a few symbols and its meanings.

Symbol	Description	Symbol	Description
A < B	A is the parent of B	A << B	A is an ancestor of B
A $ B	A and B are sisters	A $+ B	B next sister of A
A <i B	B is i^{th} child of A	A <: B	B is only child of A
A <<# B	A on head path of B	A <<- B	B is rightmost desc.
A .. B	A precedes B in depth-first traversal of tree		
A <<C B	A dominates B via unbroken chain of Cs		

Figure 3.8: Simple syntax in Tregex

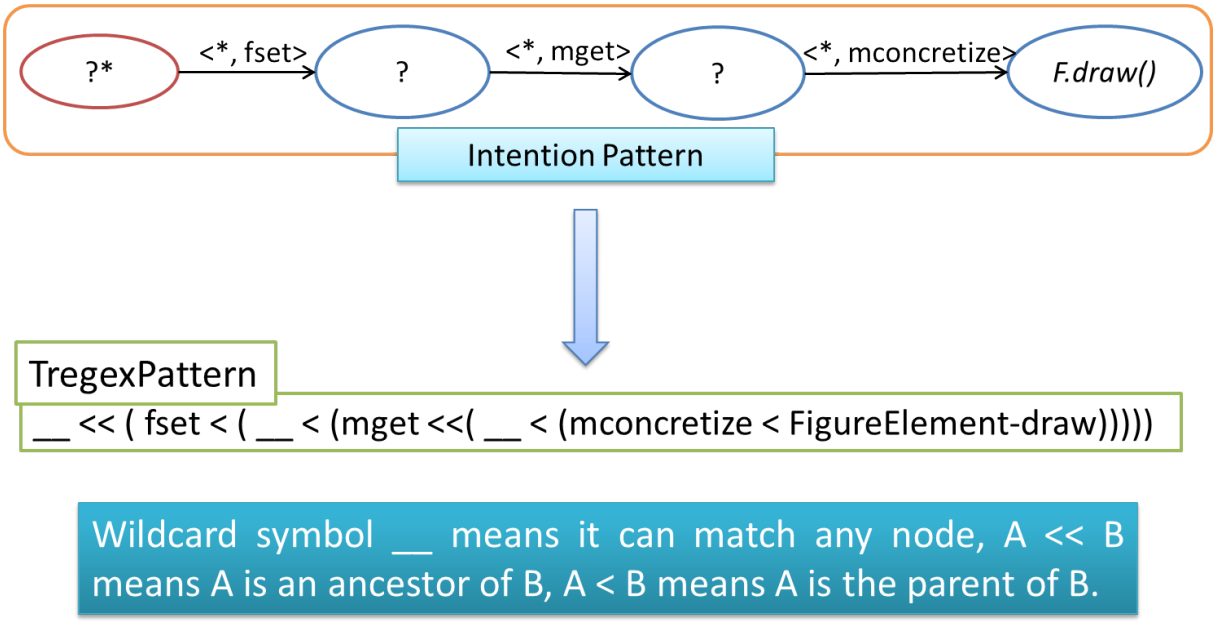

Figure 3.9: An example of Tregex Pattern

EXAMPLE 3.4. *Figure 3.9 shows an example of Tregex Pattern in figure editor system. It describes two different forms of one pattern.*

3.5 Code generator

Code generator generates an analysis-based pointcut for each specified pointcut automatically.

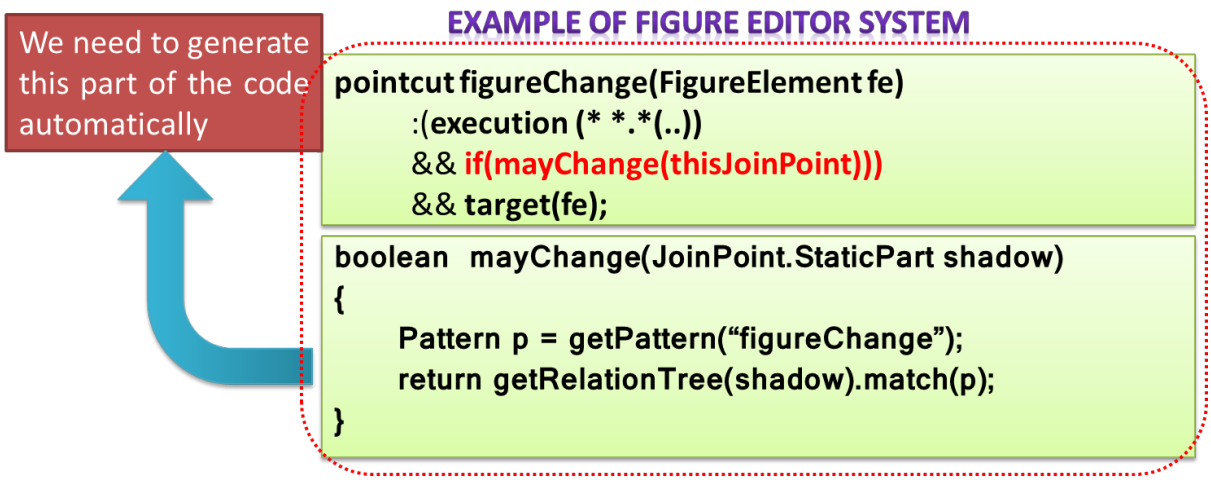

Figure 3.10: An example of code generator

3.5.1 Generate Analysis-based Pointcut Automatically

Code generator is done by filling string templates. Figure 3.10 shows an example of code which generated by this component automatically. The pattern object represents the intention pattern of a certain pointcut. The pointcut matches the method execution join points when a figure object changes its visual properties. The method `mayChange` takes a join point and return true if the method changes the visual properties of a figure object. The methods `getPattern`, `getRelationTree` are defined in our library. StringTemplate(`http://www.stringtemplate.org/`) is a java template engine for generating source code, web pages, emails, or any other formatted text output. In order to generate the analysis-based pointcut, we need to define a code-template. Figure 3.11 shows an example of a code template in figure editor system.

```
$\n$
  pointcut $pt_name$($args; separator=","$)$\n$ : $statement$ $\n$
      && !within($class$) $\n$
      && if($condition_name$(thisPointPoint)) $\n$;

  static boolean $condition_name$(JoinPoint jp){
    String jp_name = jp.getSignature().toString().replaceAll(" ","");
    return getRelationTree(jp).match(p);
  } $\n$
```

Figure 3.11: Example of code template

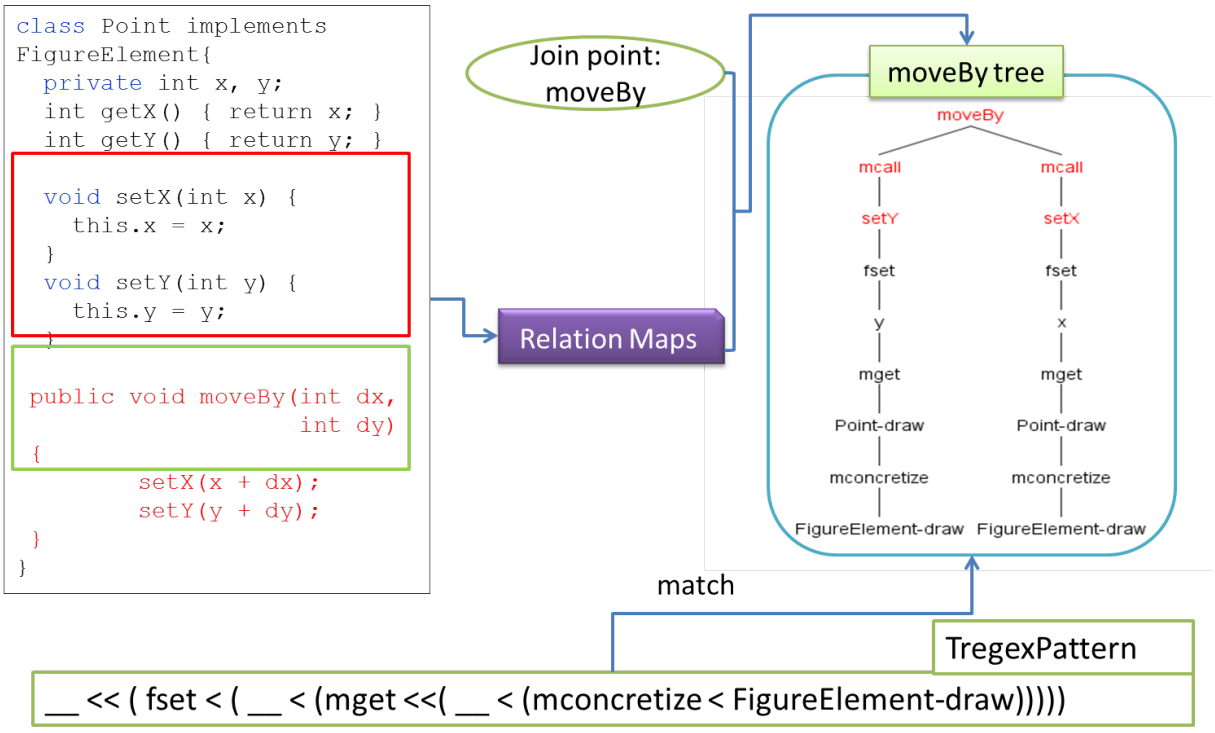

Figure 3.12: Procedure of match join point trees in figure editor system

3.5.2 Match join point trees by pattern

This section explains how to match join point trees by intention pattern. To a given join point shadow, we can obtain its corresponding element in relation maps, use these elements as roots to generate relationship trees from the relation maps. The root in each trees is the given join point shadow. After that we use intention pattern to match the relationship tree. Figure 3.12 shows the procedure of how to match join point trees by pattern. In this example, we need to explain that method `moveBy` whether can be matched by analysis-based pointcut is equivalent to whether it can be matched by the pattern. moveBy tree is generated by using join point shadows from relation maps on the left of the Figure 3.12. TregexPattern is extracted by the pattern generator before which we have explained in section 3.4.2. We use a Tregex tool to verify whether moveBy tree can be matched by the intention pattern. Figure 3.13 shows the result of the experiment. In this figure, we can see that "1 unique trees found with 6 total matches". In fact this tool checks sub-trees, so there are six sub-trees are matched by the pattern. However, in our framework, we merely use Tregex API to get a boolean value in order to verify whether the join point tree can be matched by an intention pattern.

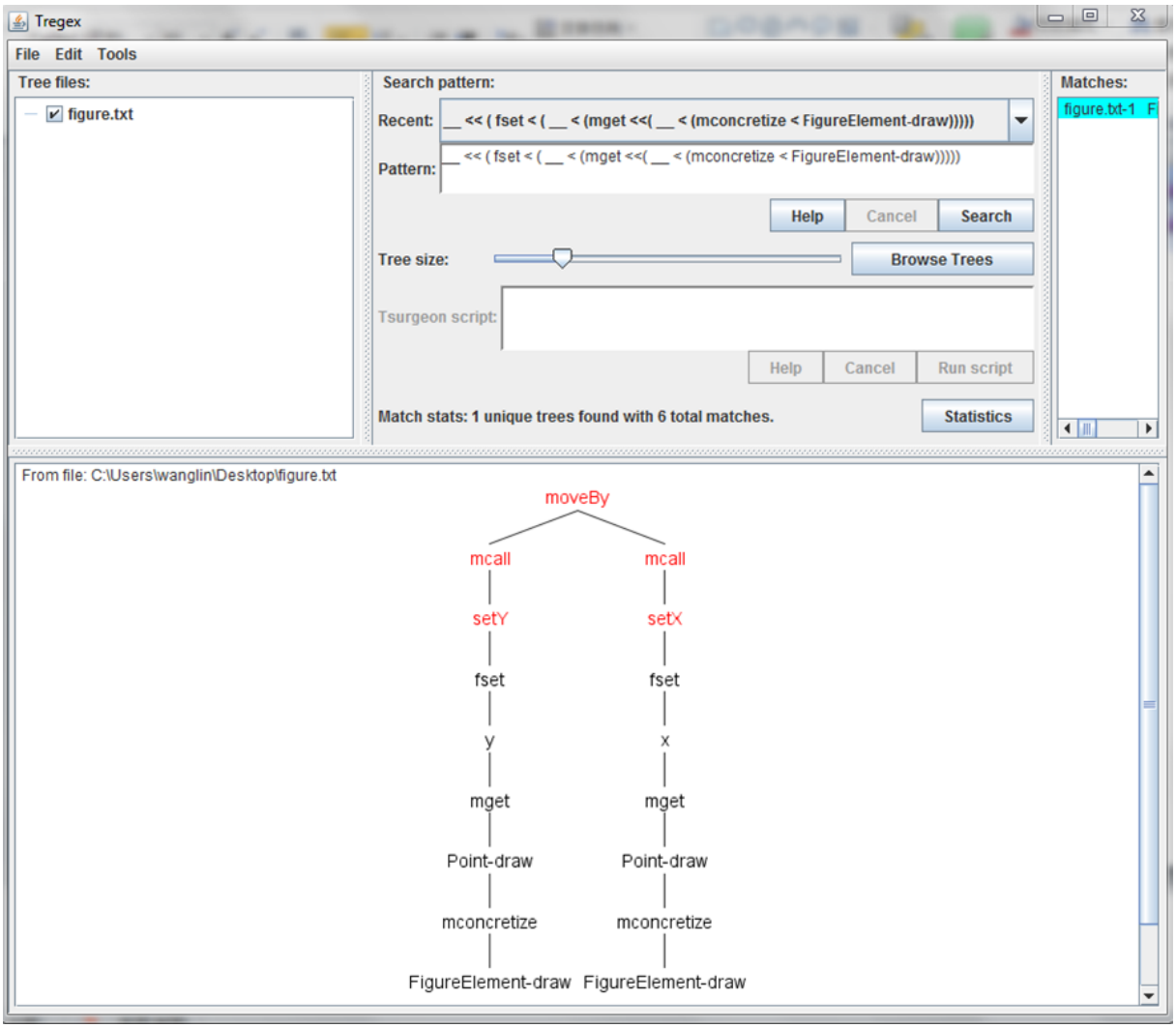

Figure 3.13: Match result in figure editor system

3.6 Stable interfaces/names

We presume that stable interfaces/names are used to define the target. For instance, in the example of figure editor system, the method `FigureElement.draw` is one of the most important method and changing the name of this method is very unlikey. Because method `FigureElement.draw` is defined in an abstract interface. So when add some new functions to the original system, we rarely change the method in an interface. If we modify abstract method, all the method in sub-classes should be changed. Therefore, changing the name of this method is unlikely. On the other hand, we often use some other Java APIs or frameworks(e.g., Java persistence framework) in our software to implement some specific functions. The methods in these APIs or frameworks are rarely to change, thus we presume that the names of these methods are stable names. Consequently, we recommend programmer using Java stable names/interfaces to define their target in order to make

their aspect-oriented program more robust by using our framework.

Chapter 4

Case Study and Evaluation

4.1 Description

We have implemented Nataly in Java, and evaluate our approach by using seven scenarios. We check whether the pointcuts need to be changed if the base program is changed in seven ways. We assume that the classes `Point`, `Line` and interface `FigureElement` in Figure 2.4 and the advice in Figure 2.5 are in the initial version of the target program.

4.2 Case study Scenarios and Evaluation

We compare the original name-based pointcut and analysis-based pointcut in seven different scenarios which proposed by Ostermann et al.[21] to the classic Figure Editor System. Table 4.1 shows each of the program evolution scenarios. Name-based pointcut means the pointcut of `change`, analysis-based pointcut is generated from name-based pointcut. The mark "`-`" means pointcut breaks, and "`+`" means piontcut works well.

Scenario	Change	Name-based pointcut	Analysis-based pointcut
SC1	(Class definition change). Inserting a new color field(should change display) to the class Point and a correspondent setter method	+	+
SC2	(Class definition change) Inserting a new method moveBy(should change display) to the class Point	-	+
SC3	(Class definition change) Inserting a new date field(should not change display) to the class Point and a correspondent setter method	-	+
SC4	(Class definition change) Renaming method setX from class Point to changeX.	-	+
SC5	(Class hierarchy change) Inserting a new class into the FigureElement hierarchy (Such as Circle)	+/-	+
SC6	(Object graph change) Use an object of Pair to store the coordinates of a Point and a correspondent setter method	+	+
SC7	(Control flow change) Inserting a new enable field to the class Point to control if an object should be exhibited on the display or not.	-	-

Table 4.1: Comparison of robustness of pointcuts, based on original name-based pointcut and analysis-based pointcut

Scenario	Change	Name-base PointCcut	Analysis-based PointCut
SC1	(Class definition change) Inserting a new ***color*** field(should change display) to the class Point and a correspondent setter method.	+	+

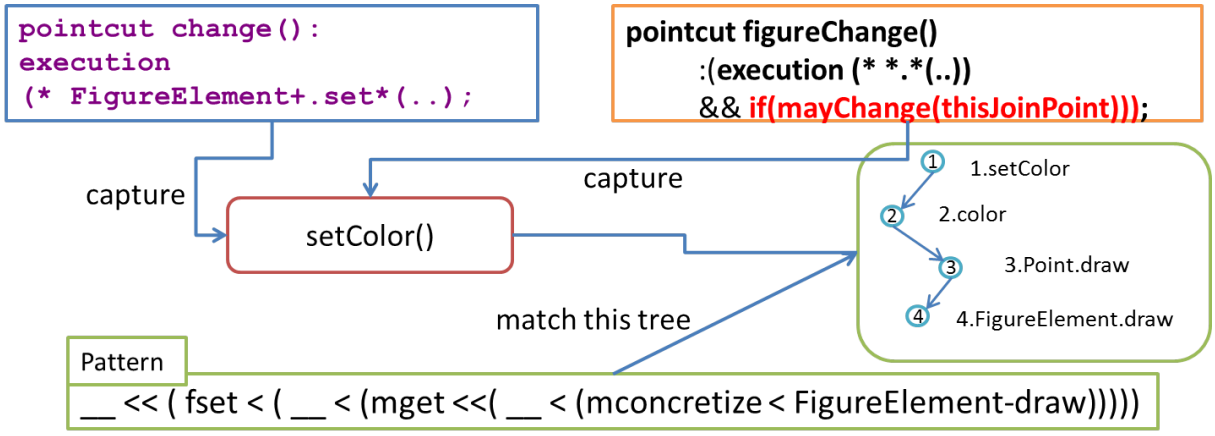

Figure 4.1: An example of senario1

Scenario	Change	Name-base PointCcut	Analysis-based PointCut
SC2	(Class definition change) Inserting a new mthod moveBy(should change display) to he class Point	-	+

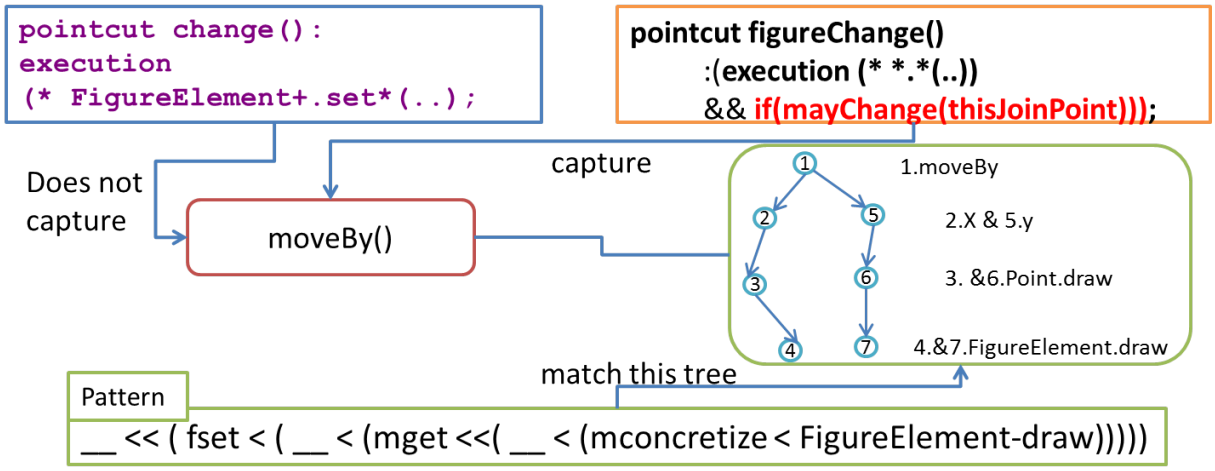

Figure 4.2: An example of senario2

SCENARIO 1. *Figure 4.1 shows an example of first scenario, In this case we add a new color field to the class* `Point` *and a corresponding setter method. Name-based pointcuts require that this method starts with text* `set`*. In this case, we add a new method called* `setColor`*, thus the solution based on name-based pointcut works well as robust(+). On the other hand, its analysis-based counterpart also works well. Even though new color field does not exist in the intention pattern, but the tree of* `setColor` *can be matched by intention pattern. The tree of* `setColor` *is as follows: setColor* $\xrightarrow{fset}$ *color* $\xrightarrow{mget}$ *Point.draw* $\xrightarrow{mconcretize}$ *FigureElement.draw. Thus,the method* `setColor` *can be captured by analysis-based pointcut.*

SCENARIO 2. *Figure 4.2 shows an example of second scenario. In this case, we add a new method moveBy to the class* `Point`*. Name-based pointcut requires that this method starts with text* `set`*. However the name of method* `moveBy` *does not contain text* `set`*, thus, the solution based on name-based pointcut breaks(-). On the other hand, its analysis-based counterpart works well. Because the tree of* `moveBy` *can be matched by intention pattern. The tree of* `moveBy` *is as follows: moveBy* $\xrightarrow{fset}$ *x* $\xrightarrow{mget}$ *Point.draw* $\xrightarrow{mconcretize}$ *FigureElement.draw is one of the branches, moveBy* $\xrightarrow{fset}$ *y* $\xrightarrow{mget}$ *Point.draw* $\xrightarrow{mconcretize}$ *FigureElement.draw is another branch. Thus, the method* `moveBy` *can be captured by analysis-based pointcut.*

Scenario	Change	Name-base PointCcut	Analysis-based PointCut
SC3	(Class definition change) Inserting a new date field(should not change display) to the class Point and a correspondent setter method	-	+

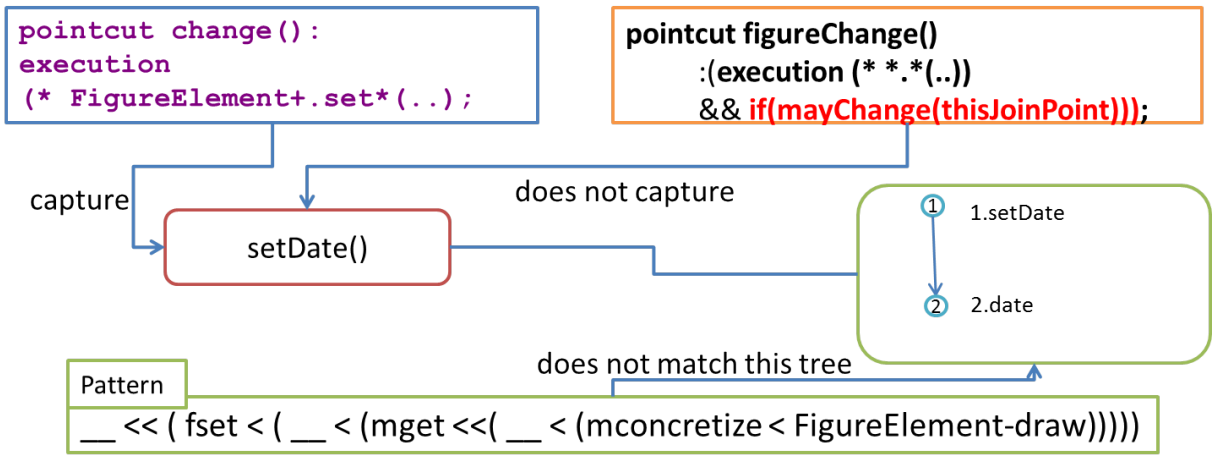

Figure 4.3: An example of senario3

SCENARIO 3. *Figure 4.3 shows an example of third scenario. In this case, add a field date which does not modify the display state into class* `Point`*, and a corresponding setter method. The name-based pointcut requires that this method starts with text* `set`*. In this case, we add a new method called* `setDate`*, thus, the name-based pointcut will capture this method when the value of the field* `date` *changed, and redraw the screen. However, this method should not be captured. Thus the solution based on name-based pointcut breaks(-). On the other hand, its analysis-based counterpart works well. Because the tree of setDate cannot be matched by intention pattern. The tree of* `setDate` *is as follows: setDate* $\xrightarrow{fset}$ *date. Thus, the method setDate cannot be captured by analysis-based pointcut.*

Scenario	Change	Name-base PointCcut	Analysis-based PointCut
SC4	(Class definition change) Renaming method setX from class Point to changeX.	-	+

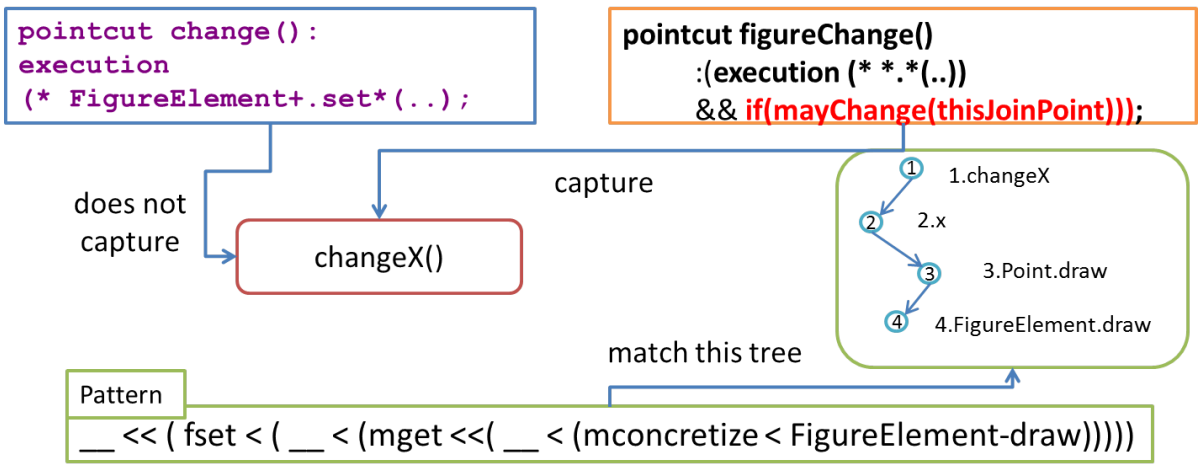

Figure 4.4: An example of senario4

SCENARIO 4. *Figure 4.4 shows an example of fourth scenario. In this case, method setX from class Point is renamed to changeX. The name-based pointcuts requires that this method starts with text* `set`*. Thus, the name-based pointcut cannot capture the new method changeX, the solution based on name-based pointcut cannot work well, it is not robust(-). On the other hand, its analysis-based counterpart works well. Because the tree of changeX can be matched by intention pattern. The tree of changeX is as follows: changeX* $\xrightarrow{fset}$ *x* $\xrightarrow{mget}$ *Point.draw* $\xrightarrow{mconcretize}$ *FigureElement.draw. Thus,the method changeX can be captured by analysis-based pointcut.*

Scenario	Change	Name-base PointCcut	Analysis-based PointCut
SC5	(Class hierarchy change) Inserting a new class into the FigureElement hierarchy (Such as Circle)	+/-	+

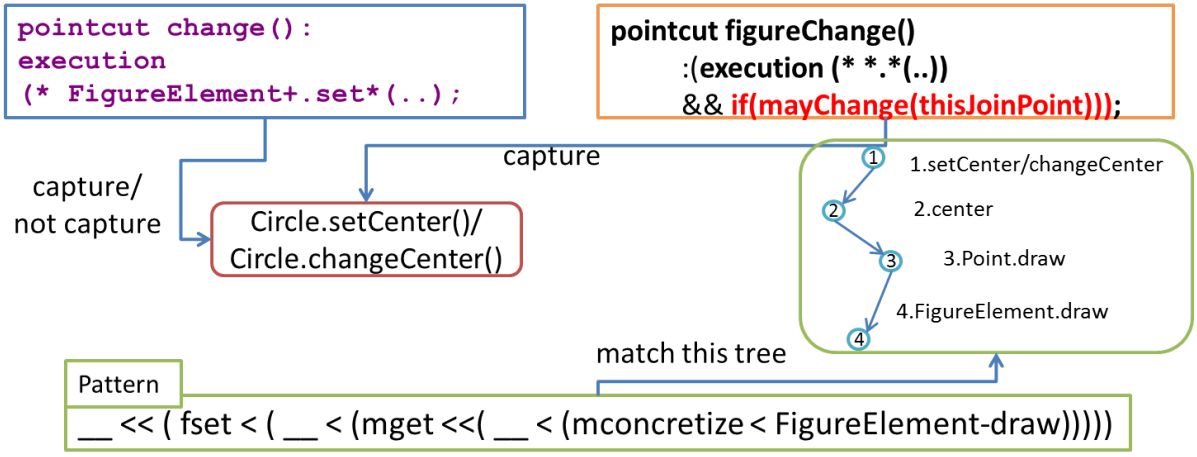

Figure 4.5: An example of senario5

SCENARIO 5. *Figure 4.5 shows an example of fifth scenario. In this case, a new class* `Circle` *is inserted into the* `FigureElement` *hierarchy. In such scenario, the name-based pointcut may be robust. Because it depends on the name of the method in class* `Circle`*, if setter methods of visual fields start with* `set` *and non-visual setter methods do not start with* `set`*, the solution based on name-based pointcut robust. Otherwise, it breaks. On the other hand, its analysis-based counterpart is robust. Because if the new method changes visual properties of a figure object, then the tree of this method will be matched by intention pattern. The tree of* `changeCenter` *is as follows:* $changeCenter \xrightarrow{fset} center \xrightarrow{mget} Point.draw \xrightarrow{mconcretize} FigureElement.draw$*. Thus, the method* `changeCenter` *can be captured by analysis-based pointcut. Otherwise, other methods in* `Circle` *class which do not change the value of visual properties that cannot be matched by pattern. and cannot be captured by analysis-based pointcut. Therefore, the analysis-based pointcut is robust(+).*

Scenario	Change	Name-base PointCcut	Analysis-based PointCut
SC6	(Object graph change) Use an object of Pair to store the coordinates of a Point and a correspondent setter method	+	+

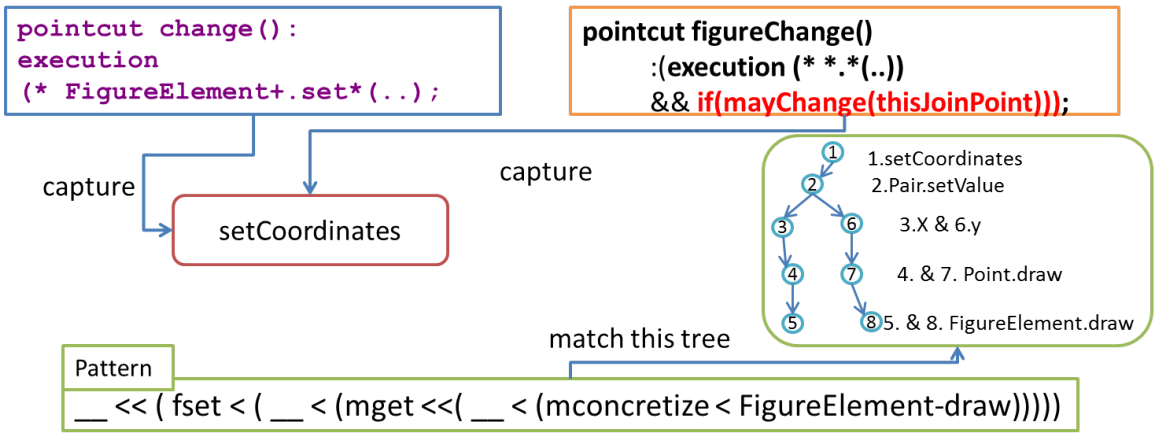

Figure 4.6: An example of senario6

Scenario	Change	Name-base PointCcut	Analysis-based PointCut
SC7	(Control flow change) Inserting a new enable field to the class Point to control if an object should be exhibited on the display or not.	-	-

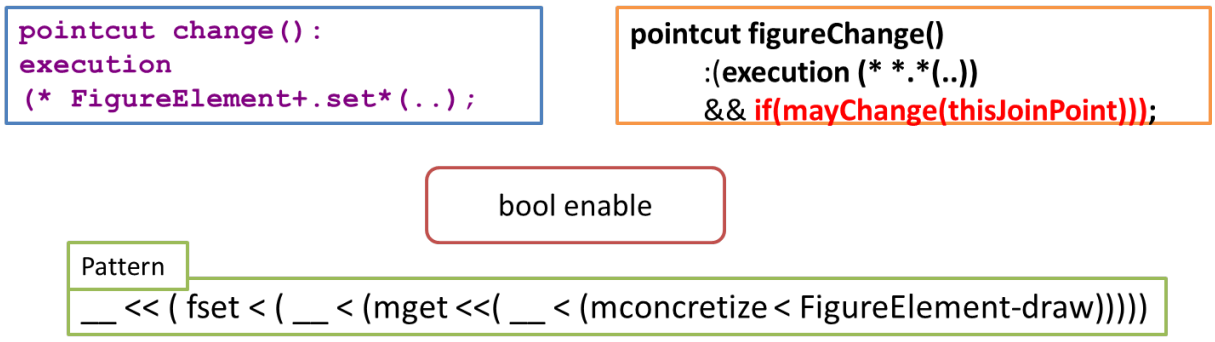

Figure 4.7: An example of senario7

SCENARIO 6. *Figure 4.6 shows an example of sixth scenario. In this case, a field of Pair is used to store the coordinates of a* `Point`*, instead of using two int filed (*`x` *and* `y`*), and add a corresponding setter method namely,* `setCoordinates`*. Class* `Pair` *has two fields*

`x` and `y`, The name-based pointcut requires that this method starts with text `set`. Thus, it can capture new method `setCoordinates`, and it is robust(+). On the other hand, its analysis-based counterpart also can work well. Because the tree of `setCoordinates` can be matched by intention pattern. The tree of `setCoordinates` is as follows: setCoordinates $\xrightarrow{mcall}$ *Pair.setValue* $\xrightarrow{fset}$ *x* $\xrightarrow{mget}$ *Point.draw* $\xrightarrow{mconcretize}$ *FigureElement.draw is one of the branches, and setCoordinates* $\xrightarrow{mcall}$ *Pair.setValue* $\xrightarrow{fset}$ *y* $\xrightarrow{mget}$ *Point.draw* $\xrightarrow{mconcretize}$ *FigureElement.draw is another branch. Thus `setCoordinates` can be captured by analysis-based pointcut.*

SCENARIO 7. *Figure 4.7 shows an example of last scenario. In this case, a new boolean filed controls if a figure object is enabled or not. Only enabled figures are allowed drawn on the screen. Therefore, in this case, we need to modify the change pointcut signature and also need to change the `FigureElement` interface. As result, neither of these two pointcut is robust.*

4.3 Evaluation

Pointcut	+	-	+/-	Rate of robustness
Name-based pointcuts	2	1	4	28.6%
Analysis-based pointcuts	6	0	1	85.7%

Table 4.2: Results for the seven evaluated change scenarios

Due to the complexity of new pointcut languages, they are difficult to be written by developers. However, our approach can generate analysis-based pointcut automatically. Moreover we use seven change scenarios to evaluate robustness of analysis-based pointcut and original name-based pointcut. The results are shown in Table 4.2. The performances of analysis-based pointcuts are obviously better than the original ones. The implementation based on name-based pointcut has presented lower degree of robustness than its counterpart. In fact, it was robust to two of seven considered scenarios. On the other hand, the solution based on the analysis-based pointcut which generated by our framework only not robust to one of seven changes. Exceptions to this control flow change, our analysis-based pointcuts are robust to all other change scenarios.

Chapter 5

Conclusion and Future Work

This dissertation proposed Nataly, a framework to translate name-based pointcuts into robust analysis-based pointcuts automatically. Nataly consist of three significant components, namely, relation analyzer, pattern generator and code generator. Our framework uses names of name-based pointcuts and source code as input, Relation analyzer generates eleven relation maps, and these relation maps are separated into two types, adjacent relation map and opposite relation map. Pattern generator creates relationship trees by using relation maps and join point shadows. Intention pattern is extracted from relationship trees and persisted to Tregex format. Finally code generator generates code of analysis-based pointcut by using StringTemplate in our implementation. We evaluate our approach by using a case study of seven possible changes. It shows that the generated analysis-based pointcuts are more robust than their name-based counterparts against seven program changes, however they still break if the program changes get complicated. Frankly, we have to declare that our approach can alleviate the problem of fragility, however it cannot solve all the possible instances of fragile pointcut problems currently.

Our approach has several contributions:

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

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

- Our implementation of framework is excellent integration with AspectJ. Analysis-

based pointcut in our approach merely relies on existing AOP constructs such as conditional pointcuts. Moreover, the advanced compiler SCoPE[6] can support our analysis-based pointcut.

Potential future work entails increase accuracy of join points which captured by our analysis-based pointcut. First, Integrating other approaches proposed by several related works may be is one direction. For example using confidence[16] as a degree to filter the captured join points for more accuracy. Second, our approach attempts to approximate the essence of a programmer's intentions, so adapt some natural language processing techniques may be another direction of our future work. Third, in the future, we also need to support pointcut designators as an input in order to support anonymous pointcut trasformation. Once we have obtained a more powerful intention pattern, an evaluation of its robustness with more complicated case studies is required.

Appendix A

AspectJ Syntax Guide

AspectJ is an aspect-oriented extension to Java. The language is fully compatible with pure Java. However, it introduces new kinds of structures and new keywords to write aspects. This appendix presents a summary of the syntax of AspectJ.

AspectJ adds to Java just one new concept, a join point and thats really just a name for an existing Java concept. It adds to Java only a few new constructs: pointcuts, advice, inter-type declarations and aspects. Pointcuts and advice dynamically affect program flow, inter-type declarations statically affect a programs class hierarchy, and aspects encapsulate these new constructs. A join point is a well-defined point in the program flow. A pointcut picks out certain join points and values at those points. A piece of advice is code that is executed when a join point is reached. These are the dynamic parts of AspectJ. AspectJ also has different kinds of inter-type declarations that allow the programmer to modify a programs static structure, namely, the members of its classes and the relationship between classes. AspectJs aspects are the unit of modularity for crosscutting concerns. They behave somewhat like Java classes, but may also include pointcuts, advice and inter-type declarations[1].

A.1 General structure of aspects

In AspectJ, aspects are syntactically similar to Java classes. Aspects are defined via the aspect keyword, where class would have been used to define a class in Java. Aspects can contain several categories of members:

- Java classes, methods, and fields, in the same way as they would be contained in a class;
- inter-type declarations (ITD) (also know as introductions) make it possible to intervene in the structure of other classes or aspects
- pointcut descriptors: these can be named and are formed by combinations of conjunctions and disjunctions of pointcut expressionsincluding primitive pointcuts

- pieces of advice: before, after or around pieces of advice can be considered as the aspect equivalents of methods. They do not have names, but they contain a pointcut (named or anonymous). They contain the Java instructions to execute when encountering join points matched by their pointcut;
- declarations

A.2 Inter-type declarations

Inter-type declarations make it possible to add new members (fields and methods) to classes, A basic example is shown in Figure A.1. Without aspect ToStringAspect, class Test does not override method toString() defined in Object. Aspect ToStringAspect introduces method Test.toString() into class Test. This is a modification of the structure of the class that is visible throughout the system.

```
1  public class Test {
2    int value = 10 ;
3    public static void main(String[] args) {
4      Test t = new Test() ;
5       System.out.println("Test: "+t) ;
6    }
7  }
8  public aspect ToStringAspect {
9    public String Test.toString() {
10     return Integer.toString(value) ;
11   }
12 }
```

Figure A.1: Inter-type declaration example.

A.3 Pointcut descriptors

The following is an extract from the AspectJ programming guide.

A pointcut is a program element that picks out join points and exposes data from the execution context of those join points. Pointcuts are used primarily by advice. They can be composed with boolean operators to build up other pointcuts. The primitive pointcuts and combinators provided by the language are:

- call(MethodPattern) Picks out each method call join point whose signature matches MethodPattern.
- execution(MethodPattern) Picks out each method execution join point whose signature matches MethodPattern.

- get(FieldPattern) Picks out each field reference join point whose signature matches FieldPattern.
- set(FieldPattern) Picks out each field set join point whose signature matches FieldPattern.
- call(ConstructorPattern) Picks out each constructor call join point whose signature matches ConstructorPattern.
- execution(ConstructorPattern) Picks out each constructor execution join point whose signature matches ConstructorPattern.
- initialization(ConstructorPattern) Picks out each object initialization join point whose signature matches ConstructorPattern.
- preinitialization(ConstructorPattern) Picks out each object pre-initialization join point whose signature matches ConstructorPattern.
- staticinitialization(TypePattern) Picks out each static initializer execution join point whose signature matches TypePattern.
- handler(TypePattern) Picks out each exception handler join point whose signature matches TypePattern.
- adviceexecution() Picks out all advice execution join points.
- within(TypePattern) Picks out each join point where the executing code is defined in a type matched by TypePattern.
- withincode(MethodPattern) Picks out each join point where the executing code is defined in a method whose signature matches MethodPattern.
- withincode(ConstructorPattern) Picks out each join point where the executing code is defined in a constructor whose signature matches ConstructorPattern.
- cflow(Pointcut) Picks out each join point in the control flow of any join point P picked out by Pointcut, including P itself.
- cflowbelow(Pointcut) Picks out each join point in the control flow of any join point P picked out by Pointcut, but not P itself.
- this(Type or Id) Picks out each join point where the currently executing object (the object bound to this) is an instance of Type, or of the type of the identifier Id (which must be bound in the enclosing advice or pointcut definition). Will not match any join points from static contexts.

- target(Type or Id) Picks out each join point where the target object (the object on which a call or field operation is applied to) is an instance of Type, or of the type of the identifier Id (which must be bound in the enclosing advice or pointcut definition). Will not match any calls, gets, or sets of static members.
- args(Type or Id, ...) Picks out each join point where the arguments are instances of a type of the appropriate type pattern or identifier.
- PointcutId(TypePattern or Id, ...) Picks out each join point that is picked out by the user-defined pointcut designator named by PointcutId.
- if(BooleanExpression) Picks out each join point where the boolean expression evaluates to true. The boolean expression used can only access static members, parameters exposed by the enclosing pointcut or advice, and thisJoinPoint forms. In particular, it cannot call non-static methods on the aspect or use return values or exceptions exposed by after advice.

Pointcut0 && Pointcut1 Picks out each join point that is picked out by both Pointcut0 and Pointcut1.

Pointcut0 || Pointcut1 Picks out each join point that is picked out by either pointcuts. Pointcut0 or Pointcut1.

A.3.1 Pointcut definition

Pointcuts are defined and named by the programmer with the pointcut declaration.

```
pointcut publicIntCall(int i):
    call(public * *(int)) && args(i);
```

A named pointcut may be defined in either a class or aspect, and is treated as a member of the class or aspect where it is found. As a member, it may have an access modifier such as public or private.

```
class C {
  pointcut publicCall(int i):
  call(public * *(int)) && args(i);
}
class D {
  pointcut myPublicCall(int i):
  C.publicCall(i) && within(SomeType);
}
```

Pointcuts that are not final may be declared abstract, and defined without a body. Abstract pointcuts may only be declared within abstract aspects.

```
abstract aspect A {
  abstract pointcut publicCall(int i);
}
```

In such a case, an extending aspect may override the abstract pointcut.

```
aspect B extends A {
  pointcut publicCall(int i): call(public Foo.m(int)) && args(i);
}
```

For completeness, a pointcut with a declaration may be declared final.

Though named pointcut declarations appear somewhat like method declarations, and can be overridden in subaspects, they cannot be overloaded. It is an error for two pointcuts to be named with the same name in the same class or aspect declaration.

The scope of a named pointcut is the enclosing class declaration. This is different than the scope of other members; the scope of other members is the enclosing class body. This means that the following code is legal:

```
aspect B percflow(publicCall()) {
    pointcut publicCall(): call(public Foo.m(int));
}
```

A.3.2 Context exposure

Pointcuts have an interface; they expose some parts of the execution context of the join points they pick out. For example, the PublicIntCall above exposes the first argument from the receptions of all public unary integer methods. This context is exposed by providing typed formal parameters to named pointcuts and advice, like the formal parameters of a Java method. These formal parameters are bound by name matching.

On the right-hand side of advice or pointcut declarations, in certain pointcut designators, a Java identifier is allowed in place of a type or collection of types. The pointcut designators that allow this are this, target, and args. In all such cases, using an identifier rather than a type does two things. First, it selects join points as based on the type of the formal parameter. So the pointcut "pointcut intArg(int i): args(i);" picks out join points where an int (or a byte, short, or char; anything assignable to an int) is being passed as an argument. Second, though, it makes the value of that argument available to the enclosing advice or pointcut.

Values can be exposed from named pointcuts as well, so

```
pointcut publicCall(int x): call(public *.*(int)) && intArg(x);
pointcut intArg(int i): args(i);
```

is a legal way to pick out all calls to public methods accepting an int argument, and exposing that argument.

There is one special case for this kind of exposure. Exposing an argument of type Object will also match primitive typed arguments, and expose a "boxed" version of the primitive. So,

```
pointcut publicCall(): call(public *.*(..)) && args(Object);
```

will pick out all unary methods that take, as their only argument, subtypes of Object (i.e., not primitive types like int), but

```
pointcut publicCall(Object o): call(public *.*(..)) && args(o);
```

will pick out all unary methods that take any argument: And if the argument was an int, then the value passed to advice will be of type java.lang.Integer.

The "boxing" of the primitive value is based on the original primitive type. So in the following program

```
public class InstanceOf {
  public static void main(String[] args) {
    doInt(5);
  }
  static void doInt(int i) { }
}
aspect IntToLong {
  pointcut el(long l) :
    execution(* doInt(..)) && args(l);
  before(Object o) : el(o) {
    System.out.println(o.getClass());
  }
}
```

The pointcut will match and expose the integer argument, but it will expose it as an Integer, not a Long.

A.3.3 Primitive pointcuts

Method-related pointcuts

AspectJ provides two primitive pointcut designators designed to capture method call and execution join points.

- call(MethodPattern)
- execution(MethodPattern)

Field-related pointcuts

AspectJ provides two primitive pointcut designators designed to capture field reference and set join points:

- get(FieldPattern)
- set(FieldPattern)

All set join points are treated as having one argument, the value the field is being set to, so at a set join point, that value can be accessed with an args pointcut. So an aspect guarding a static integer variable x declared in type T might be written as

```
aspect GuardedX {
  static final int MAX_CHANGE = 100;
  before(int newval): set(static int T.x) && args(newval) {
    if (Math.abs(newval - T.x) > MAX_CHANGE)
    throw new RuntimeException();
    }
}
```

Object creation-related pointcuts

AspectJ provides primitive pointcut designators designed to capture the initializer execution join points of objects.

- call(ConstructorPattern)
- execution(ConstructorPattern)
- initialization(ConstructorPattern)
- preinitialization(ConstructorPattern)

Class initialization-related pointcuts

AspectJ provides one primitive pointcut designator to pick out static initializer execution join points.

- staticinitialization(TypePattern)

Exception handler execution-related pointcuts

AspectJ provides one primitive pointcut designator to capture execution of exception handlers:

- handler(TypePattern)

All handler join points are treated as having one argument, the value of the exception being handled. That value can be accessed with an args pointcut. So an aspect used to put FooException objects into some normal form before they are handled could be written as

```
aspect NormalizeFooException {
  before(FooException e): handler(FooException) && args(e) {
    e.normalize();
  }
}
```

Advice execution-related pointcuts

AspectJ provides one primitive pointcut designator to capture execution of advice

- adviceexecution()

This can be used, for example, to filter out any join point in the control flow of advice from a particular aspect.

State-based pointcuts

Many concerns cut across the dynamic times when an object of a particular type is executing, being operated on, or being passed around. AspectJ provides primitive pointcuts that capture join points at these times. These pointcuts use the dynamic types of their objects to pick out join points. They may also be used to expose the objects used for discrimination.

- this(Type or Id)
- target(Type or Id)

The this pointcut picks out each join point where the currently executing object (the object bound to this) is an instance of a particular type. The target pointcut picks out each join point where the target object (the object on which a method is called or a field is accessed) is an instance of a particular type. Note that target should be understood to be the object the current join point is transfering control to. This means that the target object is the same as the current object at a method execution join point, for example, but may be different at a method call join point.

- args(Type or Id or "..", ...)

The args pointcut picks out each join point where the arguments are instances of some types. Each element in the comma-separated list is one of four things. If it is a type name, then the argument in that position must be an instance of that type. If it is an identifier, then that identifier must be bound in the enclosing advice or pointcut declaration, and so the argument in that position must be an instance of the type of the identifier (or of any type if the identifier is typed to Object). If it is the "*" wildcard, then any argument will match, and if it is the special wildcard "..", then any number of arguments will match, just like in signature patterns. So the pointcut will pick out all join points where the first argument is an int and the last is a String.

Control flow-based pointcuts

Some concerns cut across the control flow of the program. The cflow and cflowbelow primitive pointcut designators capture join points based on control flow.

- cflow(Pointcut)

- cflowbelow(Pointcut)

The cflow pointcut picks out all join points that occur between entry and exit of each join point P picked out by Pointcut, including P itself. Hence, it picks out the join points in the control flow of the join points picked out by Pointcut.

The cflowbelow pointcut picks out all join points that occur between entry and exit of each join point P picked out by Pointcut, but not including P itself. Hence, it picks out the join points below the control flow of the join points picked out by Pointcut.

Program text-based pointcuts

While many concerns cut across the runtime structure of the program, some must deal with the lexical structure. AspectJ allows aspects to pick out join points based on where their associated code is defined.

- within(TypePattern)
- withincode(MethodPattern)
- withincode(ConstructorPattern)

The within pointcut picks out each join point where the code executing is defined in the declaration of one of the types in TypePattern. This includes the class initialization, object initialization, and method and constructor execution join points for the type, as well as any join points associated with the statements and expressions of the type. It also includes any join points that are associated with code in a types nested types, and that types default constructor, if there is one.

The withincode pointcuts picks out each join point where the code executing is defined in the declaration of a particular method or constructor. This includes the method or constructor execution join point as well as any join points associated with the statements and expressions of the method or constructor. It also includes any join points that are associated with code in a method or constructors local or anonymous types.

Expression-based pointcuts

- if(BooleanExpression)

The if pointcut picks out join points based on a dynamic property. Its syntax takes an expression, which must evaluate to a boolean true or false. Within this expression, the thisJoinPoint object is available.

Note that the order of evaluation for pointcut expression components at a join point is undefined. Writing if pointcuts that have side-effects is considered bad style and may also lead to potentially confusing or even changing behavior with regard to when or if the test code will run.

A.3.4 Signatures

One very important property of a join point is its signature, which is used by many of AspectJs pointcut designators to select particular join points.

Methods

Join points associated with methods typically have method signatures, consisting of a method name, parameter types, return type, the types of the declared (checked) exceptions, and some type that the method could be called on (below called the "qualifying type").

At a method call join point, the signature is a method signature whose qualifying type is the static type used to access the method. This means that the signature for the join point created from the call ((Integer)i).toString() is different than that for the call ((Object)i).toString(), even if i is the same variable.

At a method execution join point, the signature is a method signature whose qualifying type is the declaring type of the method.

Fields

Join points associated with fields typically have field signatures, consisting of a field name and a field type. A field reference join point has such a signature, and no parameters. A field set join point has such a signature, but has a has a single parameter whose type is the same as the field type.

Constructors

Join points associated with constructors typically have constructor signatures, consisting of a parameter types, the types of the declared (checked) exceptions, and the declaring type.

At a constructor call join point, the signature is the constructor signature of the called constructor. At a constructor execution join point, the signature is the constructor signature of the currently executing constructor.

At object initialization and pre-initialization join points, the signature is the constructor signature for the constructor that started this initialization: the first constructor entered during this types initialization of this object.

Others

At a handler execution join point, the signature is composed of the exception type that the handler handles.

At an advice execution join point, the signature is composed of the aspect type, the parameter types of the advice, the return type (void for all but around advice) and the types of the declared (checked) exceptions.

A.4 Advice

The following is an extract from the AspectJ programming guide.

Advice defines pieces of aspect implementation that execute at well-defined points in the execution of the program. Those points can be given either by named pointcuts (like the ones youve seen above) or by anonymous pointcuts. Here is an example of an advice on a named pointcut:

```
pointcut setter(Point p1, int newval): target(p1) && args(newval)
  (call(void setX(int) || call(void setY(int)));
  before(Point p1, int newval): setter(p1, newval) {
    System.out.println("About to set something in " + p1 + " to the new value " + newval
  }
```

And here is exactly the same example, but using an anonymous pointcut:

```
before(Point p1, int newval): target(p1) && args(newval)
     (call(void setX(int)) || call(void setY(int))) {
        System.out.println("About to set something in " + p1 + " to the new value " + ne
}
```

Here are examples of the different advice: This before advice runs just before the join points picked out by the (anonymous) pointcut:

```
before(Point p, int x): target(p) && args(x) && call(void setX(int)) {
  if (!p.assertX(x)) return;
}
```

This after advice runs just after each join point picked out by the (anonymous) pointcut, regardless of whether it returns normally or throws an exception:

```
after(Point p, int x): target(p) && args(x) && call(void setX(int)) {
  if (!p.assertX(x)) throw new PostConditionViolation();
}
```

This after returning advice runs just after each join point picked out by the (anonymous) pointcut, but only if it returns normally. The return value can be accessed, and is named x here. After the advice runs, the return value is returned:

```
after(Point p) returning(int x): target(p) && call(int getX()) {
  System.out.println("Returning int value " + x + " for p = " + p);
}
```

This after throwing advice runs just after each join point picked out by the (anonymous) pointcut, but only when it throws an exception of type Exception. Here the exception value can be accessed with the name e. The advice re-raises the exception after its done:

```
after() throwing(Exception e): target(Point) && call(void setX(int)) {
  System.out.println(e);
}
```

This around advice traps the execution of the join point; it runs instead of the join point. The original action associated with the join point can be invoked through the special proceed call:

```
void around(Point p, int x): target(p)
    && args(x)
    && call(void setX(int)) {
    if (p.assertX(x)) proceed(p, x);
    p.releaseResources();
}
```

Appendix B

Tregex Pattern Syntax Guide

Tregex is a Tgrep2-style utility for matching patterns in trees. It can be run in a graphical user interface, from the command line using the TregexPattern main method, or used programmatically in java code via the TregexPattern, TregexMatcher and TregexPatternCompiler classes.

Tregex Pattern Syntax

Using a Tregex pattern, you can find only those trees that match the pattern you're looking for. The following table shows the symbols that are allowed in the pattern, and below there is more information about using these patterns.

List of Symbols and Meanings:

$A << B$
: A dominates B

$A >> B$
: A is dominated by B

$A < B$
: A immediately dominates B

$A < B$

$A > B$
: A is immediately dominated by B

$A\$B$
: A is a sister of B (and not equal to B)

$A..B$
: A precedes B

$A.B$
: A immediately precedes B

$A,,B$
: A follows B

A,B
: A immediately follows B

$A <<, B$
: B is a leftmost descendent of A

$A << -B$
: B is a rightmost descendent of A

$A >>, B$
: A is a leftmost descendent of B

$A >> -B$
: A is a rightmost descendent of B

$A <, B$
: B is the first child of A

$A >, B$
: A is the first child of B

$A < -B$
: B is the last child of A

$A > -B$
: A is the last child of B

$A <$ ‘B
: B is the last child of A

$A >$ ‘B
: A is the last child of B

$A < iB$
: B is the ith child of A (i ¿ 0)

$A > iB$
: A is the ith child of B (i ¿ 0)

$A < -iB$
: B is the ith-to-last child of A (i ¿ 0)

$A > -iB$
A is the ith-to-last child of B (i ¿ 0)

$A <: B$
B is the only child of A

$A >: B$
A is the only child of B

$A <<: B$
A dominates B via an unbroken chain (length ¿ 0) of unary local trees

$A >>: B$
A is dominated by B via an unbroken chain (length ¿ 0) of unary local tree

$A\$ + +B$
A is a left sister of B (same as $.. for context-free trees)

$A\$ - -B$
A is a right sister of B (same as $,, for context-free trees)

$A\$ + B$
A is the immediate left sister of B (same as $. for context-free trees)

$A\$ - B$
A is the immediate right sister of B (same as $, for context-free trees)

$A\$..B$
A is a sister of B and precedes B

$A\$, , B$
A is a sister of B and follows B

$A\$.B$
A is a sister of B and immediately precedes B

$A\$, B$
A is a sister of B and immediately follows B

$A < +(C)B$
A dominates B via an unbroken chain of (zero or more) nodes matching description C

$A > +(C)B$
A is dominated by B via an unbroken chain of (zero or more) nodes matching description C

$A. + (C)B$
: A precedes B via an unbroken chain of (zero or more) nodes matching description C

$A, +(C)B$
: A follows B via an unbroken chain of (zero or more) nodes matching description C

$A << \#B$
: B is a head of phrase A

$A >> \#B$
: A is a head of phrase B

$A < \#B$
: B is the immediate head of phrase A

$A > \#B$
: A is the immediate head of phrase B

$A == B$
: A and B are the same node

Bibliography

[1] AspectJ Web Site. http://www.eclipse.org/aspectj/.

[2] The Home of Aspect C++. http://www.aspectc.ort/.

[3] Tregex and Tsurgeon. http://nlp.stanford.edu/software/tregex.shtml.

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

[5] P. Anbalagan and T. Xie. Automated generation of pointcut mutants for testing pointcuts in AspectJ programs. In *ISSRE 2008.*, pages 239 –248, 2008.

[6] T. Aotani and H. Masuhara. SCoPE: an AspectJ compiler for supporting user-defined analysis-based pointcuts. AOSD '07, pages 161–172, 2007.

[7] N. Bhatnagar. A Survey of Aspect-Oriented Programming Languages. 2004.

[8] M. Braem, K. Gybels, A. Kellens, and W. Vanderperren. Automated pattern-based pointcut generation. In W. Löwe and M. Südholt, editors, *SC 2006*, volume 4089, pages 66–81, 2006.

[9] S. Chiba and K. Nakagawa. Josh: an open AspectJ-like language. AOSD '04, pages 102–111, 2004.

[10] V. C. Donal Lafferty. Language-Independent Aspect-Oriented Programming. 2003.

[11] V. C. Donal Lafferty. Language-Independent Aspect-Oriented Programming. 2003.

[12] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. volume 23, pages 51–60. IEEE Computer Society Press, 2006.

[13] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. AOSD '03, pages 60–69, 2003.

[14] 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.

[15] A. Kellens, K. Mens, J. Brichau, and K. Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In D. Thomas, editor, *ECOOP 2006*, volume 4067, pages 501–525, 2006.

[16] R. Khatchadourian, P. Greenwood, A. Rashid, and G. Xu. Pointcut rejuvenation: Recovering pointcut expressions in evolving aspect-oriented software. In *ASE '09*, pages 575–579, 2009.

[17] 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, pages 327–353. Springer-Verlag, 2001.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP 1997*, number 220-242, 1997.

[19] C. Koppen and M. Stoerzer. PCDiff: Attacking the fragile pointcut problem, abstract. In *European Interactive Workshop on Aspects in Software*, 2004.

[20] B. Meyer. Object-oriented software construction. 1988.

[21] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *ECOOP 2005*, pages 214–240, 2005.

[22] K. Sakurai and H. Masuhara. Test-based pointcuts: a robust pointcut mechanism based on unit test cases for software evolution. In *AOSD '08*, pages 96–107, 2007.

[23] L. Silva, S. Domingues, and M. Valente. Non-invasive and non-scattered annotations for more robust pointcuts. In *ICSM '08*, pages 67 –76, 2008.

[24] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM '05*, pages 653–656, 2005.

[25] L. Ye and K. De Volder. Tool support for understanding and diagnosing pointcut expressions. AOSD '08, pages 144–155, 2008.