Specifying Pointcuts in AspectJ

Yi Wang
Department of Computer Science
Shanghai Jiao Tong University
800 Dongchuan Rd, Shanghai, 200240, China
yi_wang@sjtu.edu.cn

Jianjun Zhao
Department of Computer Science
Shanghai Jiao Tong University
800 Dongchuan Rd, Shanghai, 200240, China
zhao-jj@cs.sjtu.edu.cn

Abstract

Program verification is a promising approach to improving program quality. To formally verify aspect-oriented programs, we have to find a way to formally specify programs written in aspect-oriented languages. Pipa is a BISL tailored to AspectJ for specifying AspectJ programs. However, Pipa has not provided specification method for pointcuts in AspectJ programs. Based on the exist work of Pipa, and related issues, this paper proposes an approach to specifying pointcuts using purity conception in JML. This paper also provides several examples to illustrate our pointcut specification approach.

1. Introduction

Aspect-Oriented Programming (AOP) [7] attempts to aid programmers in modeling the separation of concerns: breaking down of a program into distinct parts that overlap in functionality. In particular, AOP focuses on the modularization of concerns as appropriate for the host language and provides a mechanism for describing concerns that crosscut each other. AspectJ [1] [8] is an implementation of AOP for the Java programming language. Compared with common Java language, the major new constructs of AspectJ are join points, pointcuts, advice, and aspects. Join points are well-defined locations within the primary code where a concern crosscut the application. Join points can be method calls, constructor invocations, or some other points in the execution of a program. Pointcuts are constructs that match the join points, which perform a specific action called advice when triggered. Advice contains its own set of rules as to when it is to be invoked in relation to the join point that has been triggered. The aspect encapsulates above join point, pointcut, and advice.

However, the formal specification and verification of aspect-oriented programs has received comparatively little attention, although it enables the construction of more robust software. Program verification is no doubt a promising approach to improving programs' quality, because it can search all possible program executions for specific errors. To verify aspect-oriented programs, we have to find a way to formally specify programs written in aspect-oriented languages.

Pointcuts are predicates that match an event (join points) in the execution of a program. Pointcuts are modeled using expressions that identify the type, scope, or context of the events. AspectJ pointcuts provide the features of abstraction and combination, which include various designators, wildcards, and their combination with logical operators. Developers often lack high confidence on assuring that these pointcuts are specified as intended. In addition, during program evolution, specified pointcuts may not be robust enough to stand the maximum chance of continuing to match the intended join points, and only the intended join points.

Although there have several researches providing formal specifications to aspect-oriented programs, however, few studies provide pointcut specifications. The pointcut cannot be assumed to have no side effect, because there are also interactions between aspects and pointcuts. Therefore here we can consider pointcut specification with its own pre- and postconditions.

Instead of design a new language, we decided to pursuit our work under the framework of Pipa, we provide pointcut specifications for Pipa to make it more complete. Pipa is a behavioral interface specification language (BISL) [18] tailored to AspectJ. Pipa already can specify many properties for AspectJ programs efficiently. Besides designing a specification method, we should provide techniques and tools for formal specification and verification of AspectJ programs with Pipa annotations. Pipa is a natural extension of JML [10][12], so we can develop Pipa-related tools easily based the work of JML-related tools.

In this paper, we discuss a basic approach to specifying pointcuts in AspectJ. Since it is quite difficult to find solutions for all kinds of pointcuts, we only provide specification method for some mostly used pointcuts.

The rest of the paper contains following sections. Section 2 gives a brief impression of Pipa. Section 3 discusses the issues about Pointcut specifications as supplements to Pipa. Section 4 discusses related work and Section 5 concludes the paper respectively.

2. What is Pipa?

Pipa [19] is a behavioral interface specification language (BISL) tailored to AspectJ, the most common used aspect-oriented programming language. As AspectJ extends Java by adding some new concepts (join point, pointcut, introduction, advice, aspect) and associated constructs to Java, Pipa is a simple and practical extension to JML. Pipa uses the same method as JML to specify the classes and interfaces in AspectJ programs, and add some new constructs to specify aspects. Pipa is designed in such a way for following two reasons:

- ◆ Facilitating its adoption by current JML users.
- ◆ Facilitating the adoption of current JML based tool to check AspectJ programs.

Pipa provides the specification methods for aspect inheritance and crosscutting, we will introduce them respectively.

2.1 Aspect Specification

2.1.1 Advice

In Pipa, the behavior of before (after) advice can be specified by a precondition, a postcondition, and a frame condition. Around Advice is a little more complex. Pipa uses the `proceeds` *predicate* clause, to state that when control flow proceeds to the original method body (or to any additional advice if present), the around advice must make predicate hold. Pipa also uses a keyword `then`, to divide the specification of the around advice into two parts: the *before part*, which is the portion of the specification corresponding to the around advice before proceeding to the original method, and the *after part*, which is the portion corresponding to the around advice after returning from the original method but before returning to the original caller. These two specification constructs are both taken from [4].

2.1.2 Introduction

In Pipa, introduction specification mechanism is similar to method specification mechanism. Each piece of introduction may be annotated with preconditions, postconditions, and frame conditions. These preconditions, postconditions and frame conditions together form the specification of the introduction.

2.1.3 Aspect Invariant

Aspect invariants express global semantic or integrity properties for the aspect as a whole. Such invariants may involve only attributes, attributes and modules, or different modules in an aspect. An invariant of an aspect is a set of assertions (i.e., invariant clauses) that each instance of the aspect will satisfy at all times when the state is observable. Pipa uses an `invariant` clause to specify aspect invariants.

2.2 Aspect Inheritance and Crosscutting

In AspectJ, an aspect can only extend an abstract aspect, a class and implement any number of interfaces. According to these rules, Pipa use following rules in specifying aspect Inheritance:

1. A subaspect inherits the spec. of its superaspect's public and protected members and public and protected invariant;
2. A subclass inherits the spec. of its superclass's public and protected members and public and protected invariant;
3. A subaspect inherits the spec. of its superinteface's public and protected members and public and protected invariant.

AspectJ supports two kinds of crosscutting to modify the type structure and the behavior of classes an aspect crosscuts. One is structural crosscutting (by means of introduction) and the other is behavioral crosscutting (by means of advice). Pipa also has following specification rules to deal with these two cases respectively.

1. The spec. of an aspect's advice crosscuts the spec. of those classes' method that the advice crosscuts (behavior crosscutting);
2. The spec. of an aspect's introduction crosscuts the spec. of those classes' method that the introduction crosscuts (structural crosscutting).

2.3 Summary

With a few new constructs, Pipa successfully provide a framework for specification of AspectJ programs. Pipa can specify many AspectJ programs. Besides, since Pipa is a seamless extension of JML, it is easy to utilize existing JML related tools. However, Pipa does not provide mechanism for pointcut specification. We will discuss this in the following section to make Pipa more complete and expressive.

3. Pointcut Specifications

3.1 Pointcut and Its definition

Pointcuts capture, or identify, join points in the program flow. Once the join points are captured, we can specify weaving rules involving those join points—such as taking a certain action before or after the execution of the join points. In addition to matching join points, certain pointcuts can expose the context at the matched join point; the actions can then use that context to implement crosscutting functionality. There are two kinds of pointcuts, one is named, and the other is anonymous. Anonymous pointcuts are defined at the place of their usage, such as a part of advice, or at the time of the definition of another pointcut. Named pointcuts are elements that can be referenced from multiple places, making them reusable. A named pointcut defined as following:

```
[access specifier] pointcut
pointcut-name([args]) : pointcut-definition
```

AspectJ includes several primitive pointcuts, for example `call`, `get`, `set`, and so on. Programmers can compose these to define anonymous or named user-defined pointcut using and, or and not pointcut operators (‘`&&`’, ‘`||`’ and ‘`!`’).

3.2 Design Principles

A pointcut can be specified using pre- and postcondition. Informally, the pre- and postcondition can be defined in the following way:

1. The precondition of a pointcut is the properties it must hold before capture any join points. For example, consider this pointcut:

```
    call(void Point.move(int))
```

Before it capture any join points in weaving, it must satisfy that there exists a method "`void Point.move(int)`". So, the precondition is the check of the pointcut signature.

2. The postcondition of a pointcut is the properties it must hold after capture all join points in runtime. For example, consider this pointcut:

```
    call(void Point.move(int))
```

After it capture all join points, all these join points must exactly be what it should capture, neither more nor less.

In Section 3.3, we provide some method to specify pointcut's pre- and postconditions based on the above analysis. Our specification work is based on following principles:

1. If not necessary, do not add new constructs to Pipa and JML.
2. The methods should be easy to use by the researchers and developers.
3. It is no need to specify all kinds of pointcuts, the easy and mostly used ones should be considered first.

The first principle ensures we can fully utilized existed tools and extend them easily, the second bases on our objective that the specification should be useful in increase the robustness and production of practical software development, the last one reduces the complexity of our work and make our method more reasonable and efficient. In this paper, we only discuss a part of primitive pointcuts, some dynamic pointcuts such as `cflow` and `cflowbelow` need further discussions.

3.3 Pointcut Specification

As the definition of pre- and postconditions we provides in Section 3.2, we can easily find out the factors determines the pointcuts' pre- and postconditions.

Here we borrow the Purity conception from JML. In JML, the *purity* [12] is an interesting language feature. In JML, a method is *pure* either it is declared pure or it belongs to a *pure* class (interface). A method is *pure* means it can not be used in the outside of the specification. We will introduce this method in following sections according to the categories of pointcut with some simple examples. We don't cover all kinds of pointcut, we just point out how we specify most common pointcuts.

Although a pointcut can just specify a single join point in a system, the power of pointcuts comes from the economical way they match a set of join points. There are two ways that pointcut designators match join points in AspectJ. The first way captures join points based on the category to which they belong. Join points can be grouped into categories that represent the kind of join points they are, such as method call join points, method execution join points, field get join points, exception handler join points, and so forth. The pointcuts that map directly to these categories or *kinds* of exposed join points are referred as *kinded* pointcuts. The second way that pointcut designators match join points is when they are used to capture join points based on matching the circumstances under which they occur, such as control flow, lexical scope, and conditional checks. These pointcuts capture join points in any category as long as they match the prescribed condition. Some of the pointcuts of this type also allow the collection of context.

Therefore, the pointcut can be divided into the following categories, they are: *kinded pointcuts*, *control-flow based pointcuts, lexical-structure based pointcuts, execution object pointcuts, argument pointcuts,* and *conditional check pointcuts*. We will present whether our method is usable for a specific kind of pointcut and how to use it. We do not deal with the composed pointcut using pointcut operators, we will discuss this topic in the following section.

3.3.1 Kinded Pointcuts

Kinded pointcuts follow a specific syntax to capture each kind of exposed join point in AspectJ. Using the kinded pointcut can capture the join points included in one category, for example: method execution. We list the mapping of exposed join points to kinded pointcut in following table 1.

Table 1. Join point categories and corresponding kinded pointcuts.

Join Point Category	Pointcut Syntax
Method /Constructor execution	`execution(MethodSignature)` `/execution(ConstructorSignature)`
Method 、/Constructor call	`call(MethodSignature)` `/call(ConstructorSignature)`
Class initialization	`staticinitialization(TypeSignature)`
Field read/write access	`get(FieldSignature)` `/set(FieldSignature)`
Exception handler execution	`handler(TypeSignature)`
Object initialization	`Initialization (ConstructorSignature)`
Object pre-initialization	`Preinitialization (ConstructorSignature)`
Advice execution	`adviceexecution()`

Here, we use method call as an example to illustrate our method. In the first step, we define an abstract class called `S_Pointcut`, and declare it as pure as following, in this class we defined two member methods whose return value is boolean:

```
public/**@pure@*/ abstract class S_Pointcut
{......
    public abstract boolean Precheck();
    public abstract boolean Postcheck();
    ......
}
```

In the first step, according to the categories of piontcut, we implement different classes inherent above abstract class to treat different kinds of pointcut. For these methods are pure, so it can only be used in annotations.

For example, we define class *P_call* to deal with the pointcut of method call. The rule for naming is a "P" (means pointcut) connects with the type of pointcut using an underscore "_". The method *Precheck* will take charge of the checking of precondition, while the method *Postcheck* will deal with postcondition checking. The parameter of these two methods is the *name* of this pointcut in default.

```
public/*@pure@*/ class P_call extends S_Pointcut
{......
    public boolean Precheck(parameters list);
    public boolean Postcheck(parameters list);
  ......
}
```

When we specify the pointcut, we just need to use this to method as following:

```
 /**@ requires.P_call.Precheck("p");
    @ ensures P_call.Postcheck("P");
    @ */
public pointcut P() call(void Point.setX(int))
```

Using this approach, we can define classes such as *P_execution, P_get, P_set,* etc. All of them are subclass of the abstract class `S_Pointcut.` We can put our code for these classes together with the AspectJ program and specification into a package.

Our approach are suitable for all kinded pointcut, that because all these pointcut can be calculated before the real execution of the AspectJ Program. The *pure* method we defined above can calculate the set of expected join points and compare them with the join point captured during the execution time. The user can also define their own classes and methods for specifying pointcuts. Their classes also can inherit and implement the abstract class `S_Pointcut.` Besides, the users can also just write pure method belonging to the aspect for specifying pointcut, this provides great flexibility to them.

3.3.2 Control-Flow Based Pointcuts

The control-flow based pointcuts capture join points based on the control flow of join points captured by another pointcut. The control flow of a join point defines the flow of the program instructions that occur as a result of the invocation of the join point. These pointcuts included pointcut defined by keywords `cflow` and `cflowbelow.` We can not apply our approach mentioned above to specify these pointcuts. The reason is simple, the join point captured by these pointcuts are totally dynamic. It is almost impossible to try to find expected these join points before the runtime execution accurately. So, we make the decision to make our approach not support this kind of pointcuts.

3.3.3 Lexical-Structure Based Pointcuts

A lexical scope is a segment of source code. It refers to the scope of the code as it was written. Lexical-structure based pointcuts capture join points occurring inside a lexical scope of specified classes, aspects, and methods. There are two pointcuts in this category: `within()` and `withincode()`. However, the Lexical-structure based pointcuts are seldom used independently. They are always connected with other pointcut designators by pointcut operators. It has no different between the common uses of pointcut operators which we will discuss later.

3.3.4 Others

There are still several other kinds of pointcut. These are Execution object pointcuts, Argument pointcuts and Conditional check pointcuts. Execution object pointcuts match the join points based on the types of the objects at execution time. The pointcuts capture join points that match either the type `this`, which is the current object, or the `target` object, which is the object on which the method is being called. It is used in a similar way as above pointcut such as `within()` and `withincode()`. That means they are also not used solely. Besides, argument pointcuts and conditional check pointcuts also must be used with other pointcuts. So, to specify them seems meaningless.

3.3.5 Combined Pointcuts

We have pointed out that our specification method can be applied to all primitive kinded pointcuts. However, other kinds of pointcuts need to be composed with primitive kinded pointcuts. We do not consider the combinations use *cflow* and *cflowbelow*. We just discuss the combinations use pointcut operators, which are '`&&`', '`||`' and '`!`'. Generally, our specification method can be used in combination. We use following example to illustrate it.

```
/**@ requires Pipa_pointcut.Precheck("p");
   @ ensures Pipa_pointcut.Postcheck("P");
   @ */
public  pointcut  P()  call(void  Point.setX(int))
             ||execution(void Point.setX(int))
```

However, when implement the methods *Precheck* and *Postcheck*, one must be very careful to ensure the right semantics of these operators. In most cases, we can not compose the *Precheck* and *Postcheck* methods of
For example, suppose pointcut p defined in following way: `P() m|| n,` here, *m* and *n* represent a single pointcut. `Precheck("p")` dose not equal with the value of `Precheck("m")||Precheck("n")`, but equals to `Precheck("m")&&Precheck("n")`.
The postcondition is even more complicate and depends on the kinds of *m* and *n*.

3.4 An Example

Basing on above discussion, we present an example to illustrate how to integrate our pointcut specification approach into the Pipa specification. Please note that we omit the specification here. In this example, we define two pure methods whose return value is Boolean to deal with the pre- and postcondition of the pointcut "`P`".

```
package samples.pointcut.Pipa
aspect A{

public /**@pure*/ boolean Pre_p(String s);
public /**@pure*/ boolean Post_p(String s);
/**@ requires Pre_p("P");
   @ ensures Post_p("p");*/
pointcut P():
call (void classA.method(int))&& arg(x);

/**@ public behavior
   @ requires x >= MIN_X && x <= MAX_X;
   @ ensures true;
   @ signals (Exception z) false;
   @ also
   @ public behavior
   @ requires x < MIN_X || x > MAX_X;
   @ ensures false;
   @ signals (Exception z)
   @ z instanceof RuntimeException; */
before (int x) : P() {
   if ( x < MIN_X || x > MAX_X )
       throw new RuntimeException();
   }
}

Class classA{
Private int x;
    void method(int x){
         x = 5;
    }
}
```

Figure 1. An example of pointcut specification.

4. Related Work

Clifton [4][5][6] and Leavens' efforts on modular aspect-oriented reasoning require the specification of aspect advice. Their works aims to support modular aspect-oriented reasoning by extending AspectJ with new language constructs, and provide aspect categories. Their works focus on specifying advice in an aspect, not on issues about how to specify introduction, aspect invariant, specification inheritance and crosscutting, and pointcut.

Krishnamurthi et al. [9] presented an application of model checking to verifying that aspects do not violate properties verified of the base program. The technique is novel in that a base program interface may be calculated from the base program and a fixed set of pointcut descriptions (which may depend on dynamic properties). Once calculated, verification of aspects can be done without access to the base program (the base program interface and pointcut descriptions don't change).

Lorenz and Skotiniotis [13] discussed how to extend Design by Contract (DbC) to aspect-oriented programming. They pointed out that there is no generally correct execution sequence of object assertions and aspect assertions and provide a further classification of aspects as agnostic, obedient, or rebellious defines the order of assertion validation that needs to be followed.

They also described the application of this classification in a prototyped DbC tool for AOP named CONA [14].

Zhao *et al.* [19] defined Pipa, which is a Behavioral Interface Specification Language, tailored for AspectJ. Pipa implements the framework on specifying the programs written in AspectJ. Pipa extends JML, with just a few new constructs, to specify AspectJ aspects. Pipa also supports aspect specification inheritance and crosscutting. It also presents several examples of Pipa specifications, and discusses how to transform an AspectJ program together with its Pipa specification into a corresponding Java program and JML specification. Our work on pointcuts specification is based on Pipa.

5. Concluding Remarks

In this paper, we proposed an approach to specifying pointcuts in AspectJ programs. We used two method to specify the pre- and postcondition of pointcut. We also discussed related issues such as visibility and specification redundancy and provided an integrated example and discussed some related issues.

As future work, we will refine our work in two aspects. Firstly, we would like to find reasonable solutions for the pointcuts which we can not specify now, such as `cflow` and `cflowbelow`. Second, we would like to implement a support tool for Pipa, which is a crucial step for facilitating Pipa into practical use.

Acknowledgements This work was supported in part by National Natural Science Foundation of China (NSFC), Grant Number: 60673120. We would like to thank Cheng Zhang and Sai Zhang for their comments on this work.

References

[1] http://www.eclipse.org/aspectj.
[2] JML Tools: http://sourceforge.net/projects/jmlspecs.
[3] Jbossaop: http://labs.jboss.com/portal/jbossaop.
[4] Curtis Clifton and Gary T. Leavens. Observers and assistants: A proposal for modular aspect oriented reasoning. *In FOAL 2002 Proceedings: Foundations of Aspect-Oriented Languages Workshop*, pp 33–44, Enschede, the Netherlands, 2002.
[5] Curtis Clifton and Gary T. Leavens. Spectators and assistants: Enabling modular aspect-oriented reasoning. Technical Report 02-10, Iowa State University, Department of Computer Science, 2002.
[6] Curtis Clifton and Gary T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. Technical Report 03-15, Iowa State University, Department of Computer Science, 2003.
[7] G. Kiczales. *et al.* Aspect-Oriented Programming. *In Proc. of 11th European Conference on Object-Oriented Programming (ECOOP97)*, pp.220-242, LNCS, Vol.1241, Springer-Verlag, June 1997.
[8] G. Kiczales *et al*. An Overview of AspectJ. *In Proc.of 15th European Conference on Object-Oriented Programming (ECOOP01)*, pp.327-352, LNCS, Vol.2072, Springer-Verlag, June 2001.
[9] Shriram Krishnamurthi, *et al.* Verifying aspect advice modularly. *In Proc. of the 12th ACM SIGSOFT symposium on the Foundations of software engineering (FSE2004)*, Newport Beach, California, USA, 2004. ACM Press, pages 137–146.
[10] G. T. Leavens. *et al*. Jml: notations and tools supporting detailed design in java (poster session). *In Proc. of the 15th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA00)*, pages 105–106, Minneapolis, Minnesota, Oct. 15-19 2000.
[11] G T. Leavens and A L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. *In Proc. Formal Methods: World Congress on Formal Methods in the Development of Computing Systems*, Toulouse, France, September 1999. volume 1709 of Lecture Notes in Computer Science, pages 1087–1106. Springer-Verlag.
[12] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06m, Iowa State University, Department of Computer Science, Feb. 2000.
[13] D H. Lorenz and T Skotiniotis. Extending Design by Contract for Aspect Oriented Programming. Technical Report NUCCIS0414. College of Computer & Information Science Northeastern University.
[14] D H. Lorenz and T Skotiniotis. Contracts and Aspects Technical Report NU-CCIS-03-13. College of Computer and Information Science Northeastern University
[15] Olaf Spinczyk, *et al.* AspectC++: an aspect-oriented extension to the C++ programming language. *In Proc. of the 14th International Conference on Tools Pacific*, pp. 53–60. Australian Computer Society, Inc., 2002. ISBN 0-909925-88-7.
[16] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. *In Proc. of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE04)*, Newport Beach, CA, USA, 2004. ACM Press.
[17] Yang Meng Tan. Formal Specification Techniques for Engineering Modular C Programs, *volume 1 of Kluwer International Series in Software Engineering.* Kluwer Academic Publishers, Boston, 1995.
[18] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.
[19] Jianjun Zhao and Martin C. Rinard. Pipa: A behavioral interface specification language for AspectJ. *In Proc. Fundamental Approaches to Software Engineering (FASE03),* LNCS 2621, Warsaw, Poland, 2003. Springer-Verlag.