A Semantics of Pointcuts in AspectJ

Gang Xie\textsuperscript{a,b}, Bo Yang\textsuperscript{c}, Mingyi Zhang\textsuperscript{d,e} \textsuperscript{*}

\textsuperscript{a}College of Computer Science and Information, Guizhou University, Guiyang 550001, China
\textsuperscript{b}College of Mathematics and Computer Science, Guizhou Normal University, Guiyang 550001, China
\textsuperscript{c}Department of Physics and Electronic Information Science, Guiyang College, Guiyang 550001, China
\textsuperscript{d}Chongqing University of Arts and Sciences, Chongqing 400715, China
\textsuperscript{e}Guizhou Academy of Sciences, Guiyang 550001, China

Abstract

In recent years, some researchers have defined many formal semantics for aspect oriented languages. However, only several papers described the semantics of pointcut pipeiing. Unfortunately, these semantics are not complete, independent on software engineering and not easy to be understood by software designers and developers. In this paper, we propose a more simple and complete source-lever formal semantics and give some properties about the function which describes our semantics so as to support the incremental software design process. Our semantics can also deal with seventeen kind pointcuts.

© 2013 Xie Gang et al. Published by Elsevier B.V.
Selection and peer review under responsibility of Information Engineering Research Institute

Keywords. Semantics, Aspect, Aspect orientation.

\textsuperscript{*} This work is partially supported by the project funded by the science and technology department of GuiZhou Province P.R China (QianKeHeJiLKS[2012]No.35) and the National Natural Science Foundation of China under Grant No.61262029

\textsuperscript{*} Corresponding author. Tel.: +86-159-8512-3377
E-mail address: zhangmingyi045@yahoo.com.cn (Zhang Mingyi).
1. Introduction

G. Kiczales et al. [1] have developed Aspect-Oriented Programming (AOP). AOP can model crosscutting concerns of software. It has always been generating much interest in the software engineering and language communities, and in many other areas. At the same time, various AOP languages have been developed, such as AspectJ [2], AspectC++[3] and so on. Among these AOP Languages, AspectJ is the most popular one. Obviously, it is important to define the semantics which describes the AspectJ. The semantics is not only essential for programmer to express their requirement, but also useful for AspectJ compiler.

Nowadays, some researchers have defined many formal model for aspect oriented languages[4-11]. However, only several papers described the semantics of pointcut piping [4, 9, 10, 11]. Ohad Barzilay et al.[10] investigated the semantics of calling and executing pointcuts in AspectJ only. Nadia Belblidia .et al[11] presented the semantics for the static pointcut piping process in AspectJ back-end compiler. Wand et al.[4] gave a denotational semantics which handles the advice and dynamic join points. In the semantics, they dealt with eleven kinds pointcuts only. Avgustinovand[9] defi ned the semantics for static pointcuts as a set of rewriting rules from AspectJ to Datalog, a language similar to prolog. These semantics are not complete, independent on software engineering and not easy to be understood by software designers and developers. So they cannot serve to software development. In order to provide a guide for software development, a more simple and complete source-level formal semantics is required. In this paper, we propose such semantics and give some properties about the function which describes our semantics so as to support the incremental software design process. Our semantics can deal with seventeen kind pointcuts.

The rest of the paper is organized as follow: Section 2 gives the syntax of AspectJ. Section 3 defines the semantics. Section 4 discusses related work. Finally, Future works and some conclusions are reported in Section 5.

2. An Aspect-oriented Language

The language is similar to aspectJ and we have defined its syntax using the BNF. The syntax is shown in Fig. 1. It supports most of the essential aspect-oriented features, including aspects, advices, pointcuts, join points, and.

In Fig.1, x and y are variables, while the terminal is a type name such int and Boolean and represents the type of variable x when it is declared, denoted by denet(x). The terminal a is a name of attribute, m a name of method, a expression whose value is boolean, cn a primitive type constant, f an operation on a primitive data type. Any text that appears in a pair of square brackets is optional. An overlined text denote the sequence of the elements .

A program prog in AspectJ is made up of a class declarations sequence cdecls, a sequence of aspect declarations adecls and a main method Main.

A class C is declared with a visibility annotation(public, private, protected), its attributes and methods, and optionally as a direct subclass of another class D. For further details we refer the reader to [12].

An aspect A is declared with a visibility annotation, its attributes and methods, its pointcuts and advices and optionally as a direct subclass or subaspect of another class or aspect B.

A method declaration mdef, attribute declaration adef and command c are explained in [12].

A pointcut pc is declared with formal parameters T y and its pcd. A pcd is primitive a pointcut designator. AspectJ defines some primitive pointcut designators. For example, the pcd, call(MethodPattern), is used to pick out each method call join point whose signature pipeies MethodPattern. Other pcds are explained in [1]. The user can define the named or anonymous pointcut designators by composing the primitive pointcut
designators. According to the space limited, the wildcard operators (“*”, and “..”) and subtype pattern (“+”) in the MethodPattern are ignored.

A advice declaration $addef$ includes the type of advice(before, after, around) which determines how it interacts with the join points it is defined over, a formal parameter list which is same as the pointcut $pc$, a pointcut $pc$ which has been declared and a advice body(c) which is a command specifying.

**Fig. 1. Syntax of the aspectJ**
3. Semantics

3.1. Join point, pointcut

As is known, Join point, pointcut and advice are important construct. So we need define some mathematical structure for them. The definitions are given as follow.

**Definition 1.** A join point \( jop \) is a structure \(<jpt, o, id, vs, returntype>\) where:

- \( jpt \in \{fpcall, pexecution, fset, fget, init, fpreinit, aexecution, pwithin, pwithincode\} \)
- \( o \in (self \cup \Pi) \), where the self represents the current active object [12] and \( \Pi \) represents the set of objects created so far.
- \( id \in (OP \cup ATTR \cup \{new\}) \), where the \( OP \) represents the set of method names in the program, the \( ATTR \) is the set of all attributes names in the program and the \( new \) represents constructor in the program.
- \( vs \in (Expression^* \cup \{e\}) \), where the \( Expression \) represents the set of expression in the program.
- \( returntype \in (\Gamma \cup CNAME)^* \cup \{e\} \), where the \( \Gamma \) represents the set of built in primitive types, the \( CNAME \) represents the set of names of the classes declared in program and the \( e \) represents the type is void.

For example:
\(<pcall, a, new, 2, \varepsilon>\) represents that the command \( a:=new C(2) \) is a method call constructor join point.

At the same time, we use the variable \( JoinPoint \) to denote the set of join points and \( JoinPointSeq \) to denote the sequence of join points. The \( JoinPointSeq \) is defined as follows:

**Definition 2.** A sequence of join points \( JoinPointSeq \) is a n-tuple \( <jop_1, jop_2, \ldots, jop_n> \) satisfying the following conditions:

- \( n \geq 0 \). if \( n = 0 \) we denote it as \( \varepsilon \)
- \( \forall i,jop_i \in JoinPoint \)
- \( \forall i,j \text{ if } i < j, \text{ then the } jop_i \text{ is reached before the } jop_j \)
- \( head(JoinPointSeq) = jop_1 \)
- \( tail(JoinPointSeq) = jop_n \)

**Definition 3.** A pointcut \( p \) is a structure \(<A, pc, PVT, PV, pccd>\) where:

- \( A \in ASNAME \), where the \( ASNAME \) represents the set of the aspects names declared in the program.
- \( pc \in PNAME \), where the \( PNAME \) represents the set of the pointcuts names declared in the program.
- \( PVT \in (\Gamma \cup CNAME)^* \cup \{e\} \), where the \( \varepsilon \) represents the pointcut has no formal parameters.
- \( PV \in VNAME^* \cup \{e\} \), where the \( VNAME \) represents the set of the variables names declared in the program and the \( e \) represents the pointcut has no formal parameters.
- \( pccd \in PCD \), where \( PCD \) is the set of all pcds.

For example
\(<A, pc, e, e, call(C.m())>\) represents \( \text{(pointcut } pc():\text{call}(C.m())\text{)} \) is declared in aspect \( A \).

At the same time, the variable \( Pointcu \) is defined to denote the set of pointcuts.

**Definition 4**[12]. Suppose \( U \) and \( V \) are classes. \( V \subseteq^\text{as} U \) iff \( V \) is a direct superclass of \( U \) or \( V = U \).

3.2. Semantics of pcd

Because AspectJ must finish the pointcut matching process firstly in order to weave the advice into the base code, we need define the semantics for pcd to represent the pointcut piping process. The semantics is given by a function \( pipei \)

**Definition 5.** Let \( pipei \) be the function from \( PCD \) and \{\( JoinPointSeq \)\} to \{0, 1\} where:
pipei(call(T C.m(T)), < jop >) =
\[\begin{align*}
&1 \text{ if } jop.jpt = \text{pcall} \land \text{dtype}(jop.o) \leq_{su} C \\
&\land \exists N \cdot (m \in \text{op}(N) \land N \leq_{su} N \land jop.id = m) \\
&\land \text{dtype}(jop.vs) \leq_{su} T \land jop.returntype = T \\
&0 \text{ otherwise}
\end{align*}\]

pipei(execution(T C.m(T)), < jop >) =
\[\begin{align*}
&1 \text{ if } jop.jpt = \text{pexecution} \land \text{type}(jop.o) \leq_{su} C \\
&\land m \in \text{op}(C) \land jop.id = m \land \text{dtype}(jop.vs) \leq_{su} T \\
&\land \text{returntype} = T \\
&0 \text{ otherwise}
\end{align*}\]

pipei(get(T C.a), < jop >) =
\[\begin{align*}
&1 \text{ if } jop.jpt = \text{fget} \land \text{type}(jop.o) \leq_{su} C \land jop.id = a \\
&\land \text{dtype}(jop.id) = T \\
&0 \text{ otherwise}
\end{align*}\]

pipei(set(T C.a), < jop >) =
\[\begin{align*}
&1 \text{ if } jop.jpt = \text{fset} \land \text{type}(jop.o) \leq_{su} C \land jop.id = a \\
&\land \text{dtype}(jop.id) = T \\
&0 \text{ otherwise}
\end{align*}\]

pipei(initialization(C.new(T)), < jop >) =
\[\begin{align*}
&1 \text{ if } jop.jpt = \text{finit} \land \text{type}(jop.o) \leq_{su} C \\
&\land jop.id = \text{new} \land \text{dtype}(jop.vs) \leq_{su} T \\
&0 \text{ otherwise}
\end{align*}\]

pipei(preinitialization(C.new(T)), < jop >) =
\[\begin{align*}
&1 \text{ if } jop.jpt = \text{fpreinit} \land \text{type}(jop.o) \leq_{su} C \\
&\land jop.id = \text{new} \land \text{dtype}(jop.vs) \leq_{su} T \\
&0 \text{ otherwise}
\end{align*}\]

pipei(target(C), < jop >) =
\[\begin{align*}
&1 \text{ if } (jop.jpt = \text{finit} \lor jop.jpt = \text{pexecution} \lor jop.jpt = \text{fset}) \land \text{type}(jop.o) \leq_{su} C \\
&0 \text{ otherwise}
\end{align*}\]

pipei(target(id), < jop >) =
\[\begin{align*}
&1 \text{ if } (jop.jpt = \text{finit} \lor jop.jpt = \text{pexecution} \lor jop.jpt = \text{fset}) \land \text{dtype}(id) \leq_{su} C \\
&0 \text{ otherwise}
\end{align*}\]

pipei(this(C), < jop >) =
\[\begin{align*}
&1 \text{ if } (jop.jpt = \text{finit} \lor jop.jpt = \text{pexecution}) \land \text{type}(jop.o) \leq_{su} C \\
&0 \text{ otherwise}
\end{align*}\]

pipei(this(id), < jop >) =
\[\begin{align*}
&1 \text{ if } (jop.jpt = \text{finit} \lor jop.jpt = \text{pexecution}) \land \text{dtype}(id) \leq_{su} C \\
&0 \text{ otherwise}
\end{align*}\]
\[
\begin{align*}
\text{pipei}(\text{withincode}(T.C.m(\overline{t})), <jop>) &= \begin{cases} 
1 & \text{if } jop \in \text{body}(m) \land (jop.jpt = \text{pcall}) \\
0 & \text{otherwise}
\end{cases}, \\
\text{body}(m) & \text{ is the body command of the method } m[12]. \\
\text{pipei}(\text{withincode}(T.C.\text{new}(\overline{t})), <jop>) &= \begin{cases} 
1 & \text{if } jop \in \text{body}(\text{new}) \land ((jop.jpt = \text{pcall}) \\
\land jop.id \neq \text{new}) \lor jop.jpt = \text{fset} \\
\lor jop.jpt = \text{fget} \lor (jop.jpt = \text{pcontrol}) \\
\lor jop.jpt = \text{fpreinit} \\
0 & \text{otherwise}
\end{cases}.
\end{align*}
\]

\[
\begin{align*}
\text{pipei}(\text{cflow}(pcd), e) &= 0 \\
\text{pipei}(\text{cflow}(pcd), \text{JoinPoint Seq}) &= \text{pipei}(pcd, \text{head}(\text{JoinPoint Seq})) \lor \text{pipei}(pcd, \text{tail}(\text{JoinPoint Seq})) \\
\text{pipei}(pcd_1 & \& & pcd_2, <jop>) &= \text{pipei}(pcd_1, <jop>) \land \text{pipei}(pcd_2, <jop>) \\
\text{pipei}(pcd, || pcd_2, <jop>) &= \text{pipei}(pcd, <jop>) \lor \text{pipei}(pcd_2, <jop>) \\
\text{pipei}(1 pcd, <jop>) &= -\text{pipei}(pcd, <jop>)
\end{align*}
\]

The function pipei enjoys the following properties.

**Property 1 (Commutativity)**

\[
\begin{align*}
\text{pipei}(pcd_1 & \& & pcd_2, <jop>) &= \text{pipei}(pcd_2 & \& & pcd_1, <jop>) \\
\text{and} \quad \text{pipei}(pcd, || pcd_2, <jop>) &= \text{pipei}(pcd, <jop>) \lor \text{pipei}(pcd_2, <jop>)
\end{align*}
\]

**proof:**

\[
\begin{align*}
\text{pipei}(pcd_1 & \& & pcd_2, <jop>) &= \text{pipei}(pcd_1, <jop>) \land \text{pipei}(pcd_2, <jop>) \\
&= \text{pipei}(pcd_1, <jop>) \land \text{pipei}(pcd, <jcp>) = \text{pipei}(pcd_1 & \& & pcd, <jop>)
\end{align*}
\]

Similarly, \(\text{pipei}(pcd, || pcd_2, <jop>) = \text{pipei}(pcd, <jop>) \lor \text{pipei}(pcd_2, <jop>) \)

Due to limited space, the proof of the following properties will be omitted.

**Property 2 (Associativity)**

\[
\begin{align*}
\text{pipei}(pcd_1 & \& & (pcd_2 & \& & pcd_3), <jop>) &= \text{pipei}((pcd_1 & \& & (pcd_2 & \& & pcd_3), <jop>) \\
\text{and} \quad \text{pipei}((pcd, || pcd_2 & \& & pcd_3), <jop>) &= \text{pipei}((pcd, || (pcd_2 & \& & pcd_3), <jop>)
\end{align*}
\]

**Property 3 (Distributivity)**

\[
\begin{align*}
\text{pipei}((pcd_1 & \& & pcd_2), <jop>) &= \text{pipei}((pcd_1 & \& & pcd_2), <jop>) \\
\text{and} \quad \text{pipei}((pcd, || pcd_2), <jop>) &= \text{pipei}((pcd, || pcd_2), <jop>)
\end{align*}
\]

**Property 4 (Idempotent)**

\[
\begin{align*}
\text{pipei}(pcd, <jop>) &= \text{pipei}(pcd, <jop>) \quad \text{and} \quad \text{pipei}(pcd & \& & pcd, <jop>) \\
&= \text{pipei}(pcd, <jop>)
\end{align*}
\]
Property 5 (De Morgan)

\[
\text{pipei}((!(pccd_1 \& \& pccd_2), < jop >) = \text{pipei}(! pccd_1 \|| pccd_2, < jop >) \text{ and} \\
\text{pipei}((| pccd_1 \& pccd_2), < jop >) = \text{pipei}(pccd_1, \& \& ! pccd_2, < jop >)
\]

Property 6 (Law of excluded middle)

\[
\text{pipei}(pccd \| pccd, < jop >) = 1
\]

Property 7 (Law of contradiction)

\[
\text{pipei}(pccd \& \& ! pccd, < jop >) = 0
\]

Properties 6

If \(\text{pipei}(pccd_1, jop) = 1\), then \(\text{pipei}(pccd_1 \& \& pccd_2, jop) = \text{pipei}(pccd_2, jop)\)

Properties 7

If \(\text{pipei}(pccd_1, jop) = 0\), then \(\text{pipei}(pccd_1 \| pccd_2, jop) = \text{pipei}(pccd_2, jop)\)

Definition 6. For any pcd if \(\text{pipei}(pccd \& \& pccd_1, < jop >) = \text{pipei}(pccd, < jop >)\) or \(\text{pipei}(pccd \| pccd_1, < jop >) = \text{pipei}(pccd, < jop >) = 1\), then the \(pccd_1\) is a unit element of the \((\text{pipei}(PCD \times \{ \text{JoinPointSeq} \}), \& \&, \|, !)\).

For example:

\(upccd\) is a element of \(\{ pccd \|! pccd | pccd \in PCD \}\) and is also the unit element of \((\text{pipei}(PCD \times \{ \text{JoinPointSeq} \}), \& \&, \|, !)\)

Definition 6. For any pcd if \(\text{pipei}(pccd \& \& pccd_1, < jop >) = \text{pipei}(pccd_1, < jop >)\) or \(\text{pipei}(pccd \| pccd_2, < jop >) = \text{pipei}(pccd_1, < jop >)\), then the \(pccd_1\) is a zero element of the \((\text{pipei}(PCD \times \{ \text{JoinPointSeq} \}), \& \&, \|, !)\).

For example:

\(zpcd\) is a element of \(\{ pccd \& \&! pccd | pccd \in PCD \}\) and is also the zero element of \((\text{pipei}(PCD \times \{ \text{JoinPointSeq} \}), \& \&, \|, !)\)

4. Related work

The most relevant work to our work are defined in [4, 9, 11]. They define the semantics of some pointcuts.

Other frameworks presenting a formalization of the AOP paradigm are following:

Wand et al. [4] gave a denotational semantics which handles the advice and dynamic join points. The semantics is an event-based model. Lämmel[5] provided a operational semantics which defines a method-call interception through extending the object-oriented languages and the semantics is big-step. Tucker et al. [6] defines the formal model of advice and pointcut using the higher order language, Scheme. Jagadeesan et al. [7] have presented an semantics for the subset of AspectJ. But the model is quite complex so that it is difficult to prove properties of the system and understand the model for programmer.

Walker et al. [8] defined a semantics which is simpler using the lambda calculus. The semantics need translate the source-level construct in AspectJ into the calculus. As results, it is not easy to understand for programmer.
In this paper, we propose a simple and complete source-level formal semantics and give some properties about the semantics function in order to support the incremental design process of software. Our semantics can also deal with seventeen kind pointcuts.

5. Conclusion and future work

The paper describes the semantics for the following pointcuts: call, execution, get, set, initialization, preinitialization, withincode, this, target, cflow, cflowbelow and their composition. We think that the work is the beginning of defining a formal semantics of AspectJ.

We are extending the semantics described in the paper to formalize the other pointcut such as within, args and so on.

Acknowledgements

We would like to thank the referees for their thorough review. The detailed and constructive comments have helped us to bring the paper to the current form.

References