Call and Execution Semantics in AspectJ

Yishai A. Feldman
Efi Arazi School of Computer Science
The Interdisciplinary Center, Herzliya
yishai@idc.ac.il

Shmuel Tyszberowicz
Department of Computer Science
The Academic College of Tel Aviv Yaffo
tyshbe@mta.ac.il

Ohad Barzilay
School of Computer Science
Tel Aviv University
ohadbr@cs.tau.ac.il

Amiram Yehudai
School of Computer Science
Tel Aviv University
amiramy@post.tau.ac.il

Abstract

The Aspect-Oriented Programming methodology provides a means of encapsulation of crosscutting concerns in software. AspectJ is a general-purpose aspect-oriented programming language that extends Java. This paper investigates the semantics of call and execution pointcuts in AspectJ, and their interaction with inheritance. We present semantic models manifested by the current and previous releases of AspectJ, point out their shortcomings, and present alternative models.

1 Introduction

Aspect-oriented programming (AOP) [2] is a new programming paradigm that enhances object-oriented programming with the ability to modularize crosscutting concerns. These are aspects of the program, such as authentication, tracing, and logging, that affect large parts of the program across many modules. With AOP, such crosscutting concerns can be put in separate modules, which contain program fragments, called *advice*, that are inserted into other modules.

Advice can be applied to various types of events, such as object creation, method calls, access and modification of fields, and so on. These are called *join points*. Advice code can be specified to execute before or after the computation defined by a join point, or even replace it completely. Much of the power of AOP comes from the ability to specify advice that applies to multiple join points. An expression that specifies a set of join points is called a *pointcut*. This paper discusses the semantics of two types of pointcut operators in AspectJ, which is one of the most popular AOP implementations for Java.

Call and execution pointcuts in AspectJ are very similar. Both refer to a method call events, and the syntax of the pointcut expressions are identical, except for the use of the keywords `call` or `execution`. The keyword is followed by a parenthesized method signature, which specifies a set of join points. The signature can contain wildcards such as "`*`" to specify any type, "`..`" to specify any number of parameters of any type, and "`+`" to specify subclasses. For example, the pointcut `call(* A+.get*(int,..))` designates all calls to methods

- whose names start with `get`;
- that receive one `int` parameter and any number of additional parameters;
- that return any type of result; and
- that are defined in class `A` or any of its subclasses.

There are a number of other types of pointcuts; of particular interest to the issues addressed in this paper are `this` and `target` pointcuts. The former include all join points where the current object (`this`) has a certain type, and the latter include all join points where the target of the call has a given type. These pointcuts can also specify a variable to be bound to the current object or the target, respectively. (Of course, `this` pointcuts only capture join points where there is a current object, excluding, for example, the execution of static methods, and `target` pointcuts only capture non-static method-call join points, which are the only ones that have a target.) Also relevant are `if` pointcuts, which may include any boolean expression. Pointcuts can be combined using the operators `&&`, `||`, and `!`, meaning set intersection, set union, and complement, respectively.

A join point that can be captured by a call pointcut consists of the method call in the client (caller) code. A join point that can be captured by an execution pointcut consists of the whole body of the method in the supplier (callee) code. This seems to be a very subtle distinction, but it has significant ramifications. Operationally, advice to call pointcuts is inserted into the client code, while advice to execution pointcuts is inserted into supplier code. This implementation decision restricts the amount of information available to each type of advice. For example, the calling object is available in call pointcuts but not in execution pointcuts. Call pointcuts also have access to the static type of the object on which the method is invoked (the target of the call), but execution pointcuts only know its dynamic type. (There are other differences between call and execution pointcuts, but those are irrelevant to the discussion in this paper.)

From the programmer's point of view, call and execution pointcuts are very similar syntactically, and they are expected to be similar semantically as well. It turns out, however, that there are some unintuitive differences between them. In this paper we demonstrate these differences by presenting a semantic model manifested by the current (5, also called 1.5) and previous (1.1.1) releases of AspectJ, and point out their shortcomings. (We note that Jagadeesan et al. [4] mention a few of these shortcomings, but do not discuss their deficiencies.) In addition, we suggest several alternative semantics for AspectJ and investigate their expressive power and the relationships between them.

Many papers and books have been written about AOP in general, and about AspectJ in particular (e.g., [2, 3, 5]), as well as several papers giving formal semantics of simple aspect-oriented languages (e.g., [4, 6, 7, 9–11]), but none of them provides a precise (even if not completely formal) semantics of AspectJ itself. Such a semantics is necessary for language users to express their intent, and is crucial for tools that compile into AspectJ. For example, we are developing a design-by-contract [8] tool for Java. The main purpose of such a tool is to instrument the code to check assertions (method pre- and postconditions and class invariants) at run time. Existing tools we have examined perform this instrumentation in various ways, all of which have subtle errors. Our tool uses AspectJ instead of ad-hoc methods. While working on the tool, we discovered that some pointcuts we wrote did not yield the sets of join points that we expected. This has led us to conduct the study that we report on here.

We believe that a close examination of the semantics of AspectJ as manifested by the current implementation, and a discussion of the desired or "correct" semantics, is important to the AOP community. This could lead to improvements in AspectJ itself, as well as in other AOP languages. We hope that studies of the semantics of other parts of the language will follow.

We follow the approach taken by authors of the AspectJ documentation and books by ignoring implementation issues. For the purpose of this paper, we are not interested in how code instrumentation is carried out, and in the practical constraints on which classes may or may not be instrumented. We similarly ignore the implementation of the matching between pointcuts and join points in AspectJ. Instead, we treat AspectJ as a black box, and examine its behavior on carefully-chosen test cases.

2 Semantics of AspectJ

The semantics of the wildcard operators ("`*`" and "`..`") inside call and execution pointcuts are easily specified by considering them to be an abbreviation for the (infinite) union of all possible expansions. We will therefore ignore wildcards in the sequel. Also, in order to simplify the presentation, we will deal only with void functions of no arguments. This will entail no loss of generality. Because we focus in this paper on how inheritance affects the semantics of call and exeuction pointcuts, we will also ignore static methods in the sequel.

2.1 Call Semantics

We start our discussion of call semantics with the 1.1.1 release of AspectJ. As we will see later, subsequent releases have changed the semantics of call pointcuts, making them inconsistent with execution pointcuts, whose semantics has not changed.

Consider the pointcut specified by `call(void A1.f())`. This should capture all calls to the method `f` defined in class `A1`. Indeed it does, but that is due to the careful wording of the previous sentence. What happens if `f` is inherited from another class? In order to answer this question, we will consider the following hierarchy of classes:

```
public class A1
{
  public void f() { /* ... */ }
  public void g() { /* ... */ }
}

public class A2 extends A1
{
  public void h() { /* ... */ }
}

public class A3 extends A2
{
  public void f() { /* ... */ }
}
```

We then consider the following three variable definitions, in which the name of the variable indicates its static type and, if different, also its dynamic type:

```
A1 s1 = new A1();
A3 s3 = new A3();
A1 s1d3 = new A3();
```

It turns out that the pointcut `call(void A1.f())` captures the calls `s1.f()`, `s3.f()`, and `s1d3.f()`. Similarly, the pointcut `call(void A1.g())` captures the calls `s1.g()`, `s3.g()`, and `s1d3.g()`. It seems that even without the `+` subtype pattern modifier, which specifies subclasses, these pointcuts capture calls to the same method in subclasses, whether inherited or overridden. This may be a little surprising—what is the `+` modifier for, then?—but is consistent with the dynamic binding mechanism of Java. (We shall have more to say about the `+` modifier later.)

The pointcut `call(void A3.f())` captures the call `s3.f()` but not `s1d3.f()`. This implies that matching of call pointcuts is based on the static type of the variable, which is *not* consistent with the dynamic binding principle, but may perhaps be justified based on the information available at the calling point. However, the real surprise is that the pointcut `call(void A3.g())` does not capture *any* join points in our example, not even `s3.g()`! (See Figure 1 for a summary of these results.) The only difference between `f` and `g` in `A3` is that `f` is overridden whereas `g` is only inherited. Thus, it seems that for matching to succeed, it is necessary for the method to be lexically defined within the specified class—inheritance is not enough. We use the term "lexically defined" to indicate that a definition (first or overriding) of the method appears inside the definition of the class.

Thus we are led to the following model. The semantics of a pointcut will be given as a set of join points, formalized as a predicate specifying which join points are captured by the pointcut. Consider the following definitions:

- a pointcut $pc_c =$ `call(void` $C.f$`())`,

	`A1.f()`	`A3.f()`
`s1.f()`	Y	—
`s3.f()`	Y	Y
`s1d3.f()`	Y	N

	`A1.g()`	`A3.g()`
`s1.g()`	Y	—
`s3.g()`	Y	$\boxed{\text{N}}$
`s1d3.g()`	Y	N

Figure 1: Capture of join points by call pointcuts. Rows are labeled by the method pattern used in the pointcut, and rows are labeled by the actual call. Y (resp., N) in the table means that the call join point is (resp., is not) captured by the pointcut, whereas a dash means that the call is unrelated to the pointcut. The box indicates the surprising behavior.

	`A1.f()`	`A3.f()`
`s1.f()`	Y/Y	—
`s3.f()`	Y/Y	Y/Y
`s1d3.f()`	Y/Y	N/$\boxed{\text{Y}}$

	`A1.g()`	`A3.g()`
`s1.g()`	Y/Y	—
`s3.g()`	Y/Y	N/N
`s1d3.g()`	Y/Y	N/N

Figure 2: Capture of join points by call and execution pointcuts. Each cell in the table indicates capture by the call pointcut, followed by the indication for the corresponding execution pointcut.

- a variable defined as S x `= new` D`()`,
- a call join point $jp_c = x.f$`()`.

That is, the pointcut specifies a class C, and the target of the call join point has the static type S and the dynamic type D. (Obviously, D must be a descendant of S for this to compile correctly. We will denote this relationship by $D \sqsubseteq S$.) Because Java uses dynamic binding, the code that will be executed in response to the method call is the body of the method f of the class D. Note that this code is not necessarily lexically defined in the class D itself; D may inherit the implementation of f from one of its ancestors, and in this case, the execution join point jp_e refers to the code in that ancestor. With this notation, the semantics of the call pointcut is

$$jp_c \in pc_c \iff S \sqsubseteq C \ \wedge \ f \text{ is lexically defined in } C.$$

2.2 Execution Semantics

Continuing with our example, we find that call and execution pointcuts capture exactly the same method calls for `s1` and `s3`. The only difference is in the treatment of `s1d3.f()`, which is captured by both `execution(A1.f())` and `execution(A3.f())`. (Recall that it was captured by `call(A1.f())`, but *not* by `call(A3.f())`.) However, `execution(void A3.g())`, like the corresponding call pointcut, captures none of our method calls. (See Figure 2 for a summary of these results.)

We now add the definitions of the execution pointcut and join point corresponding to the call pointcut and join point above:

- an execution pointcut $pc_e =$ `execution(void` $C.f$`())`, and
- an execution join point jp_e consisting of the body of method f of class D.

The semantics of the execution pointcut now seems to be:

$$jp_e \in pc_e \iff D \sqsubseteq C \ \wedge \ f \text{ is lexically defined in } C.$$

That is, the static type is replaced by the dynamic type. Again, this can be justified by the different type information available at execution join points, but is nevertheless an inconsistency in the semantics.

	`A1.f()`	`A3.f()`	`A1+.f()`	`A3+.f()`
`s1.f()`	Y/Y	—	Y/Y	—
`s3.f()`	Y/Y	Y/Y	Y/Y	Y/Y
`s1d3.f()`	Y/Y	N/Y	Y/Y	N/Y

	`A1.g()`	`A3.g()`	`A1+.g()`	`A3+.g()`
`s1.g()`	Y/Y	—	Y/Y	—
`s3.g()`	Y/Y	N/N	Y/Y	N/N
`s1d3.g()`	Y/Y	N/N	Y/Y	N/N

	`A1.h()`	`A3.h()`	`A1+.h()`	`A3+.h()`
`s3.h()`	N/N	N/N	Y/Y	$\boxed{\text{N/N}}$

Figure 3: Capture of join points with the subtype pattern.

2.3 Subtype Pattern Semantics

The semantics of a subtype pattern such as `call(A1+.f())` should naturally be equivalent to the union of all possible expansions where `A1` is replaced by any of its descendants. This is indeed the case in AspectJ. However, because of the surprising semantics described above, this has a subtle interpretation. If

$$pc_c^+ = \texttt{call(void } C\texttt{+.}f\texttt{())}$$

is a call pointcut using subtypes, the matching rule is:

$$jp_c \in pc_c^+ \iff S \sqsubseteq C \ \wedge\ f \text{ is lexically defined in some } F \text{ s.t. } S \sqsubseteq F \sqsubseteq C.$$

In particular, the pointcut `call(A1+.h())` captures `s3.h()`, because `h` is defined in `A2`, but the same join point is *not* captured by `call(A3+.h())`, even though `A3` has this method. (See Figure 3.) This violates our expectation that `call(A3+.h())` should be a subset of `call(A1+.h())` that is identical for all join points in classes under `A3`.

Similarly, for

$$pc_e^+ = \texttt{execution(void } C\texttt{+.}f\texttt{())},$$

the matching rule is:

$$jp_e \in pc_e^+ \iff D \sqsubseteq C \ \wedge\ f \text{ is lexically defined in some } F \text{ s.t. } D \sqsubseteq F \sqsubseteq C.$$

In this case, too, the pointcut `execution(A1+.h())` captures the execution of `s3.h()`, but `execution(A3+.h())` does not.

2.4 Changes in Call Semantics

Since AspectJ version 1.2 (including version[1] 1.5.0, and also in version 1.1.1, when used with the `-1.4` compilation switch), call semantics do not require the lexical definition of the method in the class C (when `+` is not used) or one of its descendants (with the `+` modifier). Thus, call semantics are now aligned with the broad–static semantics we define in Section 3 below. However, execution semantics has *not* changed, and still requires the method to be lexically defined in C or one of its subclasses. (See Figure 4.) This inconsistency is not explained in the AspectJ documentation.

[1]AspectJ version numbers have recently been coordinated with Java version numbers. AspectJ version 1.5 corresponds Java 1.5, and since the latter is also called Java 5, the former is also called AspectJ 5.

	`A1.f()`	`A3.f()`	`A1+.f()`	`A3+.f()`
`s1.f()`	Y/Y	—	Y/Y	—
`s3.f()`	Y/Y	Y/Y	Y/Y	Y/Y
`s1d3.f()`	Y/Y	N/Y	Y/Y	N/Y

	`A1.g()`	`A3.g()`	`A1+.g()`	`A3+.g()`
`s1.g()`	Y/Y	—	Y/Y	—
`s3.g()`	Y/Y	$\boxed{\text{Y}}$/N	Y/Y	$\boxed{\text{Y}}$/N
`s1d3.g()`	Y/Y	N/N	Y/Y	N/N

	`A1.h()`	`A3.h()`	`A1+.h()`	`A3+.h()`
`s3.h()`	N/N	$\boxed{\text{Y}}$/N	Y/Y	$\boxed{\text{Y}}$/N

Figure 4: Capture of join points in AspectJ version 1.2 and above. Boxes indicate the differences from version 1.1.1 (Figure 3).

A recent document titled "The AspectJ 5 Development Kit Developer's Notebook" found on the AspectJ website (Dec. 2005) explains the semantics of call and execution pointcuts in terms of *join-point signatures*. (This device is a result of changes in the Java 1.5 specification.) Each call join point may have several signatures, depending on the class it is part of, regardless of whether the class redefines the method or not. According to this document, "[j]oin point signatures for execution join points are defined in a similar manner to signatures for call join points." The document contains no formal definition, but the section ends with the statement that "[t]here is one signature for each type that provides its own declaration of the method." The associated example makes it clear that classes not containing a lexical definition of the method do not contribute execution join-point signatures, while still contributing the corresponding call join-points signatures.

We conjecture that this difference is due to the way that execution advice is instrumented by AspectJ. The compiler inserts advice for execution join points into the method itself. Advice for `execution(void C+.f())` will be inserted into the code for the method in all subclasses of *C* that implement it. Advice for `execution(void C.f())` will be instrumented in the same way, but only if *C* itself has an implementation of the method. However, subclasses that do not implement the method will not be instrumented. In order to have execution pointcuts consistent with the new behavior of call pointcuts, it would be necessary for the compiler to insert an implementation of the method `f` into the class *C* if it does not already have one; this implementation will only call `super.f()`. This implementation of `f` will then serve as the point at which advice can be inserted. This is a small change, and is scheduled for a future release of the AspectBench Compiler for AspectJ [1].

Since version 1.2, AspectJ issues warnings (labelled `Xlint:unmatchedSuperTypeInCall`) when it finds call pointcuts that do not capture some calls because of type mismatches in the inheritance hierarchy. In our example, it issues eight warnings. Two are for the call `s1.f()` with the pointcuts `call(void A3.f())` and `call(void A3+.f())`. Two others are analogous for the call `s1.g()`. The other four are the same, but for the variable `s1d3`. In this case, the warnings might be useful, since the programmer might expect call pointcuts to have a dynamic semantics (that is, one that uses the dynamic type of the object; see Section 3). For `s1`, however, there is no distinction between the static and dynamic types, and the warnings are spurious. These warnings are probably caused by the limited inference capability of the AspectJ compiler to detect this fact.

2.5 `super` Calls

Java allows calls to overridden methods using the `super` keyword. Such calls have a corresponding execution join point, but no call join point. The existence of the execution join point is inevitable, given

Variable definition:	$S\ x$ `= new` D`()`
Call join point:	$jp_c = x.f$`()`
Execution join point:	$jp_e =$ body of method f in D
Pointcuts:	$pc_c =$ `call(void` $C.f$`())`
	$pc_e =$ `execution(void` $C.f$`())`
	$pc_c^+ =$ `call(void` C`+.`f`())`
	$pc_e^+ =$ `execution(void` C`+.`f`())`

$$\begin{array}{lcl} jp_c \in pc_c & \iff & S \sqsubseteq C \ \wedge\ f \text{ is lexically defined in } C \\ jp_e \in pc_e & \iff & D \sqsubseteq C \ \wedge\ f \text{ is lexically defined in } C \\ jp_c \in pc_c^+ & \iff & S \sqsubseteq C \ \wedge\ f \text{ is lexically defined in some } F \text{ s.t. } S \sqsubseteq F \sqsubseteq C \\ jp_e \in pc_e^+ & \iff & D \sqsubseteq C \ \wedge\ f \text{ is lexically defined in some } F \text{ s.t. } D \sqsubseteq F \sqsubseteq C \end{array}$$

(a)

$$\begin{array}{lcl} jp_c \in pc_c & \iff & S \sqsubseteq C \ \wedge\ f \text{ exists in } C \\ jp_c \in pc_c^+ & \iff & S \sqsubseteq C \ \wedge\ f \text{ exists in } S \end{array}$$

(b)

Figure 5: Semantics of AspectJ implementation: (a) version 1.1.1; (b) changes since version 1.2.

that advice for execution join points is inserted into the body of the called method. Such advice code will be executed regardless of how the method body was called. However, there is no reason why `super` calls should not have call join points as well. The AspectJ compiler can treat such calls like any other call, since all information about the method being called is available during compilation.

It seems reasonable that `super` invocations should be treated like any other method invocation. However, this raises another issue relating to the definition of join points in AspectJ. In plain Java, the addition of an overriding method that only forwards the call to its superclass does not change the semantics of the program. In contrast, the addition of an implementation

```
void f() { super.f(); }
```

in AspectJ does change which join points are captured by (execution) pointcuts. Suppose that the implementation of `f` in our example class `A3` contains a call `super.f()`. Any advice to the pointcut `execution(void A1.f())` will be invoked twice, for the implmentation of `f` in `A3` and for the `super` call to the implementation in `A1`. If we now add the above definition also to class `A2`, the advice will be executed three times, once for each of the implementations of `f` in classes `A1`, `A2` and `A3`.

An aspect language should respect invariants of the underlying language as much as possible. In this example, the effect of making a change in the Java program only affects the aspects, not the original program. However, in a scenario where aspects are developed separately from the main Java code, independent changes made by the Java programmer could affect the behavior of the aspects in unanticipated ways.

In this particular case, it is hard to see how execution join points can be implemented efficiently without violating this invariant. However, AspectJ programmers need to take this phenomenon into account. Also, if execution join points behave in this way, it seems better to make call join points behave in the same way rather than creating another internal inconsistency in the AspectJ language.

2.6 Summary

The semantics of version 1.1.1 and later version of AspectJ are summarized in Figure 5. We use the term "f exists in C" to denote the fact that the method f exists in class C, whether or not it is lexically defined in it.

The AspectJ semantics satisfies some of our intuitive expectations but violates others. The points on which AspectJ is consistent with the intuitive semantics are:

- Pointcuts with wildcards are equivalent to the union of all possible expansions.
- Pointcuts with subtype patterns are equivalent to the union of all pointcuts with subtypes substituted for the given type.
- The semantics of execution pointcuts is based on the dynamic type of the target.

On the following points the semantics of AspectJ deviates from our intuition:

- The semantics of call pointcuts is different from that of execution pointcuts, and is determined by the static type of the target rather than the dynamic type. Since version 1.2, call and execution pointcuts also differ on the lexical definition requirement.
- Execution (and, in version 1.1.1, also call) pointcuts only capture join points for classes where the given method is lexically defined.
- As a result of this, the difference between pointcuts with or without subtype patterns is subtle and unintuitive.

It is arguable whether pointcuts without subtype patterns should capture join points in subclasses at all. On the one hand, an instance of a class is *ipso facto* considered to belong to all its superclasses; this is reflected in the syntactic restrictions on assignment and parameter passing, and in the semantics of the `instanceof` operator. On the other hand, the existence of the subtype pattern modifier seems to imply the intention that a pointcut that does not use it refer only to direct instances of the specified class.

We believe that the lexical restrictions shown in these semantic definitions were unintended; their removal would greatly simplify the semantics. Some evidence that this is not the intended semantics comes from the following quote from one of the AspectJ gurus [5, p. 79]: "The [`call(* Account.* (..))` pointcut] will pick up all the instance and static methods defined in the `Account` class *and all the parent classes in the inheritance hierarchy*" (emphasis added). This was not true in AspectJ at the time of writing (version 1.1.1), but is intuitively appealing. This *is* true for the current semantics of AspectJ, but is *not* true for execution pointcuts, where intuition requires the same treatment. Thus, even an AspectJ expert's intuition differs from the implemented semantics.

3 Alternative Semantics

If the current AspectJ semantics is inappropriate, we should propose one or more alternatives. As mentioned above, such alternatives should not restrict methods to be lexically defined in the designated class. Two questions remain:

1. Should subclasses be included when the subtype pattern modifier does not appear in the pointcut?
2. Should call and execution pointcuts capture different join points?

These issues lead to four possible definitions of the semantics (see Figure 6). We use the term "broad" for those semantics that include subclasses even when subtypes are not indicated, and "narrow" for those that do not. The term "static" denotes semantics that use the static type for call pointcuts, and "dynamic" denotes those that use the dynamic type. (Both use the dynamic type for *execution* pointcuts, for which static type information is not available.)

We can simplify these definitions even further. The condition that the method f exist in D is redundant, since if it is not met, there are no join points that are candidates for the pointcut at all. The condition that f exist in S is also redundant, under the assumption that the original program compiles sucessfully. Since S is the static type of the variable x, the call $x.f()$ will be rejected by the Java compiler unless f exists in S. Under these assumptions, Figure 6 reduces to Figure 7.

Each of the four semantics is consistent, and also reasonable. Perhaps the broad–dynamic semantics best reflects object-oriented principles, in that a reference to a class may point to elements of any of its subclasses, and the type that determines matching is the dynamic rather than static type of the variable.

	Narrow		**Broad**	
Static	$jp_c \in pc_c$	$\iff S = C \,\wedge\, f$ exists in C	$jp_c \in pc_c$	$\iff S \sqsubseteq C \,\wedge\, f$ exists in C
	$jp_e \in pc_e$	$\iff D = C \,\wedge\, f$ exists in C	$jp_e \in pc_e$	$\iff D \sqsubseteq C \,\wedge\, f$ exists in C
	$jp_c \in pc_c^+$	$\iff S \sqsubseteq C \,\wedge\, f$ exists in S	$jp_c \in pc_c^+$	$\iff S \sqsubseteq C \,\wedge\, f$ exists in S
	$jp_e \in pc_e^+$	$\iff D \sqsubseteq C \,\wedge\, f$ exists in D	$jp_e \in pc_e^+$	$\iff D \sqsubseteq C \,\wedge\, f$ exists in D
	(a)		(b)	
Dynamic	$jp_c \in pc_c$	$\iff D = C \,\wedge\, f$ exists in C	$jp_c \in pc_c$	$\iff D \sqsubseteq C \,\wedge\, f$ exists in C
	$jp_e \in pc_e$	$\iff D = C \,\wedge\, f$ exists in C	$jp_e \in pc_e$	$\iff D \sqsubseteq C \,\wedge\, f$ exists in C
	$jp_c \in pc_c^+$	$\iff D \sqsubseteq C \,\wedge\, f$ exists in D	$jp_c \in pc_c^+$	$\iff D \sqsubseteq C \,\wedge\, f$ exists in D
	$jp_e \in pc_e^+$	$\iff D \sqsubseteq C \,\wedge\, f$ exists in D	$jp_e \in pc_e^+$	$\iff D \sqsubseteq C \,\wedge\, f$ exists in D
	(c)		(d)	

Figure 6: Four possible semantics: (a) narrow–static; (b) broad–static; (c) narrow–dynamic; (d) broad–dynamic.

	Narrow		**Broad**	
Static	$jp_c \in pc_c$	$\iff S = C \,\wedge\, f$ exists in C	$jp_c \in pc_c$	$\iff S \sqsubseteq C \,\wedge\, f$ exists in C
	$jp_e \in pc_e$	$\iff D = C \,\wedge\, f$ exists in C	$jp_e \in pc_e$	$\iff D \sqsubseteq C \,\wedge\, f$ exists in C
	$jp_c \in pc_c^+$	$\iff S \sqsubseteq C$	$jp_c \in pc_c^+$	$\iff S \sqsubseteq C$
	$jp_e \in pc_e^+$	$\iff D \sqsubseteq C$	$jp_e \in pc_e^+$	$\iff D \sqsubseteq C$
	(a)		(b)	
Dynamic	$jp_c \in pc_c$	$\iff D = C \,\wedge\, f$ exists in C	$jp_c \in pc_c$	$\iff D \sqsubseteq C \,\wedge\, f$ exists in C
	$jp_e \in pc_e$	$\iff D = C \,\wedge\, f$ exists in C	$jp_e \in pc_e$	$\iff D \sqsubseteq C \,\wedge\, f$ exists in C
	$jp_c \in pc_c^+$	$\iff D \sqsubseteq C$	$jp_c \in pc_c^+$	$\iff D \sqsubseteq C$
	$jp_e \in pc_e^+$	$\iff D \sqsubseteq C$	$jp_e \in pc_e^+$	$\iff D \sqsubseteq C$
	(c)		(d)	

Figure 7: Simplified semantics from Figure 6.

If we accept that call pointcuts are instrumented in the caller's code, the broad–static semantics is appropriate (and, indeed, it is the closest to the current AspectJ semantics). However, other semantics may be easier to use if they more closely reflect the intent of AspectJ programmers.

4 Expressive Power

... WORK IN PROGRESS ...

5 A Comprehensive Set of Operators

In this section, we present a proposal for syntactic and semantic changes to AspectJ. This proposal can express all the types of pointcuts described elsewhere in this paper, and is therefore more expressive than all the other possible semantics mentioned here. However, it is perhaps *not* the most convenient for programmer use, and we therefore present it more for its theoretical value than as a practical proposal.

Looking back at Figures 5 and 6, we find several types of relationships used in the various semantics:

- The first set consists of inheritance relationships: $S = C$, $S \sqsubseteq C$, $D = C$, and $D \sqsubseteq C$.
- The second set consists of lexical definition requirements on the method f; it could be required to be lexically defined in C, in some F such that $S \sqsubseteq F \sqsubseteq C$, or in some F such that $D \sqsubseteq F \sqsubseteq C$.

- The third set consists of requirements on the existence (but not necessarily lexical definition) of the method f in the classes C, S, or D.

Not all combinations of these operators make sense. As explained in the introduction, we assume that the semantics of execution pointcuts never refers to the static class S. This is true in all the semantics presented above, and is a result of the fact that advice on execution join points is instrumented in the target class, at which point no static information about the type of the target variable is available.

It is now clear that we will be able to express all these four semantics given the ability to express the relationships from the first set plus the fact that f exists in C. The former can be achieved by adopting the new `static` operator suggested at the end of Section 4 and possibly modifying the semantics of `this` and `target`. The relationship $S = C$ would be expressed by `static(C)`, and $S \sqsubseteq C$ by `static(C+)`. (Because of the assumption that execution join points contain no static information, these are only relevant for call join points.)

As mentioned in Section 4, we can capture the dynamic relationship $D = C$ using the expression `if(x.getClass() == C.class)`, and $D \sqsubseteq C$ is easily captured by `target(C)` for call pointcuts and by `this(C)` for execution pointcuts. However, it would be more consistent with the proposed `static` pointcuts to express $D = C$ by `target(C)` for call pointcuts and by `this(C)` for execution pointcuts, and $D \sqsubseteq C$ by `target(C+)` for call pointcuts and by `this(C+)` for execution pointcuts. (Note that the syntax `this(C+)` and `target(C+)` is already supported by AspectJ, but, as far as we could find, these expressions mean the same as the corresponding expressions that do not contain the `+` modifier. The warnings mentioned at the end of Section 2.4 sometimes include the `+` and sometimes do not, indicating a possible difference, but this might be due to a simple copying of the class expression from the pointcut expression.)

We can now restrict call and execution pointcuts to specify the signature of the method, without a class, since class information can be added using the first set of operators. While this results in an orthogonal set of operators, it would lose conciseness in the most common cases. Instead, we can take the most liberal meaning of call and execution pointcuts that do contain classes. We thus take `call(void C.f())` and `execution(void C.f())` to mean $D \sqsubseteq C \ \wedge\ f$ exists in C, and `call(void C+.f())` and `execution(void C+.f())` to mean $D \sqsubseteq C$. This is just the broad-dynamic semantics of Figure 7. (There may be methods f with the same signature in classes that do not inherit from C. Since the pointcut expressions specifically mention C, these should be excluded. This is the reason for the requirement $D \sqsubseteq C$ in all cases.)

Figure 8 shows how the four types of pointcuts can be expressed in the proposed scheme of this section in order to express the meanings of the original pointcuts under the four proposed semantics.

	Narrow	**Broad**
Static	`call(void C.f()) && static(C)`	`call(void C.f()) && static(C+)`
	`execution(void C.f()) && target(C)`	`execution(void C.f())`
	`call(void C+.f()) && static(C+)`	`call(void C+.f()) && static(C+)`
	`execution(void C+.f())`	`execution(void C+.f())`
	(a)	(b)
Dynamic	`call(void C.f()) && target(C)`	`call(void C.f())`
	`execution(void C.f()) && this(C)`	`execution(void C.f())`
	`call(void C+.f())`	`call(void C+.f())`
	`execution(void C+.f())`	`execution(void C+.f())`
	(c)	(d)

Figure 8: The four pointcuts in the proposed scheme of Section 5. Each part shows how to express the four pointcuts pc_c, pc_e, pc_c^+, and pc_e^+ in the relevant semantics.

In order to be able to express the AspectJ semantics of Figure 5 we need additional operators that refer to lexical definitions. In Figure 5, the lower bounds $S \sqsubseteq F$ and $D \sqsubseteq F$ are also redundant, since

$$\begin{array}{lcl}
pc_c & = & \texttt{call(void C.f()) \&\& lexical(void C.f()) \&\& static(C+)} \\
pc_e & = & \texttt{execution(void C.f()) \&\& lexical(void C.f())} \\
pc_c^+ & = & \texttt{call(void C+.f()) \&\& lexical(void C+.f()) \&\& static(C+)} \\
pc_e^+ & = & \texttt{execution(void C+.f()) \&\& lexical(void C+.f())}
\end{array}$$

Figure 9: Expressing AspectJ 1.1.1 semantics in the proposed scheme of Section 5.

the first is required for compilation without errors, and the second is required in order for any relevant join point to exist at all. We therefore need to specify that a method is lexically defined in the class C or below it. This can be achieved by a new pointcut operator `lexical`, which takes a method pattern, like call and execution pointcuts. The semantics of `lexical(void C.f())` would be that f is lexically defined in C, and that of `lexical(void C+.f())` would be that f is lexically defined in some F such that $F \sqsubseteq C$. Figure 9 shows how to express the AspectJ 1.1.1 semantics of Figure 5 in this new scheme.

6 Conclusions

The current semantics of AspectJ has some unintuitive aspects. We have presented a number of alternative semantics, and compared their expressive power. The "right" semantics for AspectJ needs to be worked out with the user community, since it ultimately depends on how AspectJ is used in practice. While semantic changes in AspectJ may be limited by the desire for backward compatibility, this discussion is also relevant for other AOP tools, both for Java and for other languages. We hope that this paper will start a fruitful and constructive discussion on these questions.

References

[1] O. de Moor, 2005. Personal communication.

[2] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Comm. ACM*, 44(10):29–32, 2001.

[3] J. D. Gradecki and N. Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java.* John Wiley & Sons, 2003.

[4] R. Jagadeesan, A. Jeffrey, and J. Reily. A calculus of untyped aspect-oriented programs. In L. Cardelli, editor, *European Conference on Object-Oriented Programming (ECOOP 2003)*, volume 2743 of *Lecture Notes in Computer Science*, pages 54–73. Springer-Verlag, 2003.

[5] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning, 2003.

[6] R. Lämmel. A semantical approach to method-call interception. In *Proc. First Int'l Conf. Aspect-Oriented Software Development (AOSD 2002)*, pages 41–55, April 2002.

[7] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In *Ninth Int'l Workshop on Foundations of Object-Oriented Languages*, 2002.

[8] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2nd edition, 1997.

[9] D. B. Tucker and S. Krishnamurthi. A semantics for pointcuts and advice in higher-order languages. Technical Report CS-02-13, Brown University, 2003.

[10] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*, pages 127–139, August 2003.

[11] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In *Ninth Int'l Workshop on Foundations of Object-Oriented Languages*, 2002.