Flexible Pointcut Implementation: An Interpreted Approach

Ali Assaf*, Jacques Noyé**
Département informatique, École des Mines de Nantes

*Ali.Assaf@emn.fr,
**Jacques.Noye@emn.fr

Abstract. One of the main elements of an Aspect-Oriented Programming (AOP) language or framework is its *pointcut language*. A *pointcut* is a predicate which selects program execution points and determines at which points the execution should be affected by an aspect. Experimenting with AspectJ shows that two basic primitive pointcuts, `call` and `execution`, dealing with method invocation from the caller and callee standpoints, respectively, lead to confusion. This is due to a subtle interplay between the use of static and dynamic types to select execution points, dynamic lookup, and the expectation to easily select the caller and callee execution points related to the same invocation. As a result, alternative semantics have been proposed but have remained paper design.

In this article, we reconsider these various semantics in a practical way by implementing them using CALI, our *Common Aspect Language Interpreter*. This framework reuses both Java as a base language and AspectJ as a way to select the program execution points of interest. An additional interpretation layer can then be used to prototype interesting AOP variants in a full-blown environment. The paper illustrates the benefits of applying such a setting to the case of the `call` and `execution` pointcuts. We show that alternative semantics can be implemented very easily and exercised in the context of AspectJ without resorting to complex compiler technology.

1 Introduction

AspectJ is the most popular AOP language (Kiczales et al., 2001; Filman et al., 2005). It extends Java with a new type of code structuring entity, called *aspect*, which contains the definition of *pointcuts* and *advices* in addition to standard Java members like fields and methods. A *pointcut* selects execution points of the so-called *base program* where extra code (the *advice*) has to be executed. These points are called *join points*. At the user level, the properties of the pointcut language like expressiveness, obliviousness, clarity of semantics, comprehensibility, flexibility, extensibility, etc. are important issues in the design of AOP languages. At the implementation level, a *weaver* is responsible for *weaving* the selection of the join points into the base program. As soon as static information is available, for instance types, and join-point selection relies on this information, part of the selection process can be dealt with statically (at

```
public class Super {
    public void f(){
   }
}
public class Middle extends Super {
}
public class Sub extends Middle {
  public void g(){
   }
}
```

Listing 1 – *Example.*

compile- or load-time). This is the case with AspectJ, see (Hilsdale and Hugunin, 2004). The static part of the selection is based on static information obtained from the program point corresponding to the join point, the join point *shadow*. The dynamic part of the selection, based on runtime values, is implemented by a *residue* injected into the base-program bytecode by the weaver. The residue is executed at runtime to decide whether a statically-selected aspect does apply.

AspectJ offers two ways to intercept method invocation: one for intercepting the invocation at the caller side with `call(`T P`.`m`(`A`))`, and another for intercepting the execution of the method body, at the callee side, with `execution(`T $P.m(A)$`)`, where T is the return type of the method of interest, P its *declaring* type, and A the types of its arguments.

A first issue is to understand what is exactly meant by *declaring* type. A first clue is given by considering Listing 1 together with:

- the calls `new Sub().f()` and `new Sub().g()`;
- the pointcuts `call(void Middle.f())` and `call(void Middle.g())`.

With the current version of AspectJ (1.6), the only selected join point is the call `a3.f()`. The only difference between the calls is that the method `g` is initially defined in a subclass of `Middle`. We cannot just expect `call(void Middle.g())` to selects calls to `g` with receivers of type `Middle` (which is actually impossible) or `Sub`, a subtype of `Middle`, the method has also to *exist* in the declaring type `Middle`. Another clue that there is no straightforward choice of semantics is that in earlier versions of AspectJ, the call to `f` was not selected either: the method had to be *defined* in the declaring type given in the pointcut.

A second issue is to understand against which *qualifying* type, on the join point side, the declaring type is matched. It is simple in the previous example because the static and dynamic types of the receivers are the same, but it is not always the case. We shall also see that the story is different on the caller side and on the callee side with the counter-intuitive result that, in the presence of the pointcuts `call(void Middle.f())` and `execution(void Middle.f())`, a call to `Middle.f()` will be captured by the pointcut `call` but the execution of the method `f` will not be captured by the pointcut `execution`. Of course, with respect to a given call, a call join point and an execution join point are different join points: with respect to a given call, the call join point occurs within the context of the caller object, whereas the execution join point occurs within the context of the receiver. Still, the intuition is that two pointcuts

```
class Service implements Runnable {
    public void run() { ... }
    public static void main(String[] args) {
        ((Runnable) new Service()).run();
    }
}
```

Listing 2 – *Example.*

`call` and `execution` using the same declaring type should select both (Barzilay et al., 2004; Avgustinov et al., 2007).

In this paper, we revisit the semantics of the pointcuts `call` and `execution` in the current version of AspectJ (1.6) as well as their possible alternative semantics, based on the earlier work of (Barzilay et al., 2004). We disconnect from compiler-based implementation and we describe the use of *join point selectors* to implement pointcuts as part of our interpreter-based framework to prototype AOPL (CALI) (Assaf, 2008). This framework is based on the semantics of AspectJ as described by the CASB (Djoko Djoko et al., 2006). We show that, in our interpreter-based framework, the implementation of the pointcuts `call` and `execution` as join point selectors is able to directly reflect the various semantics. Moreover, moving from one semantics to the other is extremely simple.

The implementation of pointcuts using an interpreted approach facilitates the understanding of the semantics and provides an executable environment to test them and potentially improve the comprehensibility, the evolvability, and the modularity of pointcuts. More generally, such an interpreted approach is an interesting lightweight alternative, at least in an exploratory phase, to compiler-based approaches to implementing AspectJ extensions (Avgustinov et al., 2005).

The paper is structured as follows: Section 2 discusses the semantics of the pointcuts `call` and `execution`. Section 3 describes our interpreter. Section 4 displays the interpreter-based implementation of the current and the alternative semantics of the pointcuts. Section 5 concludes the paper.

2 AspectJ Semantics

The pointcuts `call` and `execution` are *signature-matching* pointcuts. All these pointcuts specify a *declaring type*. Understanding the meaning of this declaring type in relation with the join point signatures is key to understanding the semantics of the signature-matching pointcuts. This section clarifies this issue with respect to the pointcuts `call` and `execution`. We first expose the semantics of these pointcuts in the current version of AspectJ (1.6) and then consider the reasons for this semantics as well as alternatives.

2.1 Call Semantics

According to the AspectJ Programming Guide (Appendix B, Semantics) (The AspectJ Team, 2003), *when matching method-call join points, the declaring type is the static type used* *to access the method*.

This leads to the common pitfall, signalled in the AspectJ Programming Guide, that the pointcut `call(void Service.run())` does not capture the call to the method `run` of Listing 2. As explained in the AspectJ Programming Guide on this very example, this is because the declaring type `Service` given in the pointcut is a subtype of the *qualifying* type `Runnable` of the join-point signature. In case of call join point, this *qualifying* type is the *static type used to access the method*.

But what is exactly the *static type used to access the method*? Let us consider the call `((Middle)new Sub()).f()` in the context of Listing 1. Is the qualifying type of the join point `Middle`, the static type of the receiver expression, or `Super`, the type where the method `f` of the class `Middle` is defined? Experimenting with AspectJ shows that the qualifying type of the join point is actually the static type of the receiver expression.

Let us go back to Listing 2. Of course, the pointcut `call(void Runnable.run())` would capture the call to `run` because the declaring type of the pointcut and the qualifying type of the join point are then the same. But this is hardly enough to understand the semantics of the pointcut `call`. Some more analysis is necessary.

It is also interesting to consider what happens if there is no cast. In that case, the call join point for `new Service().run()` is captured by both pointcuts. If we consider the pointcut `call(void Runnable.run())`, we can see that the join-point qualifying type `Service` is a subtype of the pointcut declaring type `Runnable`. Getting a different result may come as a surprise as this variant base program is not semantically different.

A second experiment consists of replacing the method `run` by a method `myrun` in both the class `Service` and the pointcuts. In that case, the call join point `new Service().myrun()` is captured by the pointcut `call(void Service.myrun())` but not by the pointcut `call(void Runnable.myrun())`, even though the join-point qualifying type (`Service`) is a subtype of the pointcut declaring type (`Runnable`). At first sight, it may look like it is because, unlike in the initial case, the method `myrun` is not defined in `Runnable`. Further experiments would show that the problem is more precisely that the method `myrun` does not *exist* in `Runnable`, *i.e.*, it is neither defined in `Runnable` nor in any of its supertypes.

To summarize, there are two conditions for a pointcut `call(`P`.`m`())` to capture a call join point e`.`m`()`, where J is the qualifying type of the join point (the static type of e):

- $J <: P$ (J is a subtype of P);
- m exists in P.

2.2 Execution Semantics

According again to the AspectJ Programming Guide, *when matching method-execution join points, if the execution pointcut method signature specifies a declaring type, the pointcut will only match methods declared in that type, or methods that override methods declared in or inherited by that type*. This is illustrated, in the AspectJ Programming Guide, by the application of the pointcut `execution(public void Middle.*())` to the example of Listing 3, a variant of Listing 1, which is said to capture any method-execution join point for `Sub.m()`. Indeed, the method `m` in `Sub` overrides the definition of `m` inherited by the pointcut declaring type `Middle`.

```
class Super {
    protected void m() { ... }
}
class Middle extends Super {
}
class Sub extends Middle {
    public void m() { ... }
}
```

Listing 3 – *Example.*

It is actually interesting to be systematic on this example and check what happens when running `new Sub().m()`, `new Middle().m()`, and `new Super().m()`. As we have just said, the pointcut `execution(public void Middle.*())` captures a join point in the first case. It does not in the second case because, although the method `m` is inherited by `Middle`, it is not overridden in `Middle`. It does not in the third case either because the method `m` of `Super` cannot be inherited from `Middle`. The second case may look surprising if we consider that there is a (virtual) copy of the method `m` in `Middle`. It is less surprising if we look at the execution in terms of dynamic lookup. There is indeed no definition of the method `m` in `Middle` and it is the definition of the method `m` in the superclass that is executed.

To summarize, there are three conditions for a pointcut `execution(`P`.`m`())` to capture a method-execution join point m`(){}`, where *this* is of dynamic type D:

- $D <: P$;
- m exists in P;
- m is (re)defined in J such that $D <: J <: P$ and not redefined between D and J.

Here, the qualifying type of the join point J is the declaring type of the method m. As $D <: J$ these conditions can then be reformulated as follows:

- $J <: P$, where J is the join-point *qualifying type*;
- m exists in P.

We then get the same conditions as with the pointcut `call` but with a specific definition of the join-point qualifying type.

2.3 Discussion

Relating the semantics of the pointcuts `call` and `execution` With a proper definition of the *qualifying type* of the method signatures of the join points, we get syntactically homogeneous definitions of the pointcuts `call` and `execution`. Still, the definition of the qualifying type of an execution join point, though in line with dynamic lookup, is tricky and leads to the source of surprise mentioned in the introduction, originally discussed by (Barzilay et al., 2004), in the context of an earlier semantics, and rediscovered by (Avgustinov et al., 2007) while precisely defining the semantics of static pointcuts in AspectJ as datalog queries. Basically, two pointcuts `call` and `execution` using the same declaring type do not always capture both the call join point and its related execution join point.

Indeed, if we consider, in the context of Listing 3, the pointcut `call(public void Middle.*())`, a join point is captured for both `new Sub().m()` and `new Middle().m()`. As a result, in the presence of an aspect comprising both the pointcuts `call` and `execution`, a call join point followed by an execution join point is captured in the first case whereas, in the second case, only a call join point is captured.

Such a discrepancy does not occur in the alternative semantics proposed in (Barzilay et al., 2004). These semantics were proposed as a reaction to the earlier semantics of AspectJ 1.1.1. The difference with the current semantics (AspectJ 1.6) was that the method of interest had to be (re)defined in the declaring type of both the pointcuts `call` and `execution`, and the qualifying type of an execution join point was simply the dynamic type of the current object (the receiver of the call).

Basically, the proposal of (Barzilay et al., 2004) consists of:

- always considering that the method of interest *exists* in the pointcut declaring type and
- defining the qualifying type of the call join points as either the static type of the receiver or its dynamic type, the qualifying type of the execution join points remaining the dynamic type of the receiver.

The first point fixes the surprise that the pointcut `call(void Middle.f())` did not capture the call `new Sub().f()` in Listing 1. The second point defines two semantics, a *static* semantics or a *dynamic* semantics depending on the definition of the qualifying type of the call join points.

The *static* semantics corresponds to the current semantics of AspectJ with respect to method calls, but method executions are still handled differently.

Semantics of the pointcut `call` An argument against using the *dynamic* semantics, which has the advantage of being simpler to explain and reason about as only dynamic types have to be considered in the join points, is that it is less efficient and that using another primitive pointcut, `target`, makes it possible to get this semantics anyway. The efficiency issue comes from the fact that in the case of the static semantics, the matching condition $J <: P \wedge m$ exists in P can be fully evaluated statically as J is a static type. There is no need for a residue. As for using the pointcut `target`, the idea is that this pointcut makes it possible to select call join points based on the *dynamic* type of the callee. For instance, in our previous example (Listing 2), it is still possible to capture the call to the method `run` using `call(void Runnable.run())&& target(Service)`.

(Barzilay et al., 2004) note that the static semantics of `call(`$P.m$`())&& target(`P`)` is slightly different from the dynamic semantics of `call(`$P.m$`())`. Indeed, if D is the dynamic type of the join point and S its static type, we have, in the first case, the condition $S <: P \wedge D <: P$, where the first conjunct comes from the pointcut `call` and the second from the pointcut `target`. As by definition $D <: S$, this amounts to $S <: P$. This has to be compared with the condition required by the dynamic semantics: $D <: P$. Conversely, if the dynamic semantics were chosen, it could make sense to change the semantics of the pointcut `target` so that it deals with static types.

Semantics of the pointcut `execution` We have seen that a pointcut `execution(`P`.`m`())` does not capture a method-execution join point m`(){}`, where the receiver is of dynamic type J, if m is not (re)defined in the hierarchy between P and J. This may look surprising but actually corresponds to the lookup semantics. In practice, if m is only defined in a superclass of P, there is no bytecode that can be instrumented in order to implement the pointcut and transfer control to the advice if necessary apart from the bytecode in the superclass. But this would then slow down the execution of the method with some additional tests on the type of `this` to distinguish the cases when the pointcut should apply and when it should not. An alternative would be to add bytecode for the missing methods at weave time. For instance, let us consider the example in Listing 3. If we redefine `m` in `Middle`: `protected void m(){super.m()}`, then `execution(public void Middle.*())` captures a join point when executing `new Middle().m()`. Such a redefinition could be done at weave time at the bytecode level. One would then get the semantics of (Barzilay et al., 2004).

Existence of the method in the declaring type At first sight, the necessity of the condition m exists in P is also arguable. As we have previously seen in Section 1 when considering Listing 1, without this condition, the pointcut `call(void Middle.g())` would then select a call `new Sub().g()`. This has the drawback that the semantics of the pointcut `call` is not purely static any longer (which is of course not an issue in the context of the dynamic semantics of (Barzilay et al., 2004)). But the main point is that this semantics can anyway be already obtained by using, instead of a type P, the *subtype pattern* P+, which stands for P or any of its subtype.

Summary The semantics of the pointcuts `call` and `execution` is not easy to grasp. Matching uses the static type of the caller in the pointcuts `call` and the dynamic type of the callee in the pointcuts `execution`. Understanding the semantics of the pointcuts `execution` requires to reason about the behavior of dynamic lookup and is counter-intuitive when a pointcut `execution(`P`.`m`())` does not capture an execution join point whereas the corresponding call join point is captured by the pointcut `call(`P`.`m`())`.

Alternative semantics that seem easier to grasp have been proposed. In particular, the dynamic semantics of (Barzilay et al., 2004) looks attractive. Matching is always based on dynamic types and the simple condition that the method of interest should exist in the type of interest is used for both the pointcuts `call` and `execution`.

Switching to such a semantics may however require a lot of work in the current compiler-based implementations of AspectJ and result in some performance overhead, whereas the exact benefits have not yet been fully assessed. It looks interesting but does it really help in practice? Therefore, it may be worth experimenting with this approach in a more agile environment. In the following sections, we show how we can, by switching to an interpreter-based approach, easily implement the current semantics of the pointcuts `call` and `execution`, and then switch to the static and dynamic semantics of (Barzilay et al., 2004) with minimal changes.

```
abstract class JoinPointSelector {
    abstract boolean staticTest(JoinPoint jp);
    abstract boolean dynamicTest(JoinPoint jp);
}
```

Listing 4 – *The class* `JoinPointSelector`*.*

3 The AspectJ Interpreter

Our interpreter represents a good example of mixing together compilation and interpretation. The base program runs as a normal Java program on a standard JVM while aspects are partially interpreted. AspectJ aspects are translated, using a parser, to a specific structure (in Java) defined by the interpreter, and are represented at runtime by Java objects conforming to this specific structure. More concretely, each aspect is represented by an instance of a class `Aspect`, which encapsulates a list of selector/advice bindings, instances of a class `SelectorAdviceBinding`. Each selector/advice binding contains a join-point selector, instance of a class `JoinPointSelector`, and an advice, instance of a class `Advice`. A join-point selector corresponds to a pointcut. Its role is to select join points.

According to the matching semantics described in (Djoko Djoko et al., 2006; Assaf and Noyé, 2008), a join point selector selects a join point in two steps, a *static* and a *dynamic* step, implemented through the methods *staticTest* and *dynamicTest* (see Listing 4). The static test is based on static information obtained from the join point shadow. The dynamic test corresponds to the execution of the AspectJ residue and decides whether a statically-matching aspect does apply.

Two-step weaving (David et al., 2001) is used to weave the interpreted aspects into the compiled Java code:

1. The first step takes place in AspectJ at compile time, through an aspect called `Platform` (see Listing 5), which results in the instrumentation by AspectJ of all possible join points in the base program by defining a pointcut `call(* *.*(..))|| execution (* *.*(..))|| set(* *)...`;

2. The second step takes place at runtime, when the advice of the `Platform` aspect, which behaves as an *interpretation layer*, evaluates the current AspectJ join point (accessed through `thisJoinPoint`) by looking for matching aspects before executing them. The "base" Java parts of the selected advices are then again executed as plain Java code.

The interpretation layer of the `Platform` aspect consists of four subprocesses (**reify**, **match**, **order** and **mix**) similar to the subprocesses of weaving described in (Kojarski and Lorenz, 2006).

In particular, in **match**, an aspect matches the current join point if the static test of a join point selector of one of its selector advice bindings selects it. The process **order** orders the matching aspects as a list of ϕ objects[1]. Each ϕ object is basically a pair dynamic test/advice, except the last ϕ object, which is the join point wrapped as a ϕ object. The evaluation starts

[1]This is a reference to the notations used by (Djoko Djoko et al., 2006) to describe the semantics of AspectJ.

```
public aspect Platform {
  pointcut reifyBase():
    (call(* *.*(..)) || execution(* *.*(..)) || set(* *) ...)
      && !within(java..*)
      && !within(org.aspectj..*)
      && !within(dynamicaspectj..*);
  pointcut reifyAdvice(): ...
  pointcut reify(): reifyBase() || reifyAdvice();
    Object around(): reify() {
      List<Phi> phis = order(thisJoinPoint, match(thisJoinPoint));
      return mix(thisJoinPoint, phis);
    }
  public static List<Phi> match(JoinPoint jp) { ... }
  public static List<Phi> order(JoinPoint jp, List<Phi> phis) {
    List<Phi> orderedPhis = aspectJOrder(phis);
    orderedPhis.add(new PhiJP(jp));
    return orderedPhis;
  }
  public static Object mix(JoinPoint jp, List<Phi> phis) {
    Phi phi = phis.remove(0);
    ...
    Object o = phi.eval(jp);
    ...
    return o;
  }
}
```

Listing 5 – *The aspect* `Platform`*.*

then with the first ϕ object. The dynamic test is performed. In the case of a positive result, the corresponding advice is executed, otherwise a special aspect-level instruction, `proceed`, is executed and the execution proceeds with the next ϕ object. More details on the aspect `Platform` can be found in (Assaf and Noyé, 2008) and the source code can be downloaded from (Assaf, 2008).

4 Implementation

4.1 Background

AspectJ provides a special reference variable, `thisJoinPoint`, accessible from the body of any advice, that contains reflective information about the current join point. The advice of the aspect `Platform` passes this information to all the reified aspect selectors, which can then access this information for matching. In the following, we will use:

- `String JoinPoint.getKind()` returns a string representing the *kind* of the join point. In particular, it returns "method−call" and "method−execution" for method-call and method-execution join points, respectively (there are also constructor calls and executions, which requires to distinguish, for instance, between a method call and a constructor call) ;

- `Signature JoinPoint.getSignature()` returns the signature of the join point. For method-call join points, it returns the signature of the method *existing* in the static type. For method-execution join points, it returns the signature of the method being executed.

We will also use the reflection API of Java, in particular:

- `Class Class.asSubClass(Class c)` casts `this` to `c` and throws an exception if `this` is not a subclass of `c`;

- `Method Class.getMethod(String name, Class[] parameterTypes)` returns the instance of `Method` (inherited or defined) in `this` with the name and parameter types given as parameters. An exception `NoSuchMethodException` is thrown if no such method is found, an exception `NullPointerException` if the value of `name` is `null`, and an exception `SecurityException` in case of a security exception in the presence of a security manager.

4.2 Call

Listing 6 gives the implementation of the selector for method calls as a subclass of `JoinPointSelector`. All the information necessary to the selector in order to determine whether a join point matches or not is actually static (it can be obtained from the join point shadow). As a result, the conditions for matching are implemented in its method `staticTest` (line 19), whereas its method `dynamicTest` (line 38) always returns true.

The implementation makes use of two boolean variables: $exists$ and $basicMatch$.

- The boolean variable `exists` is set to true (line 13) if the method exists in the pointcut declaring class `pointcutClass`. This directly corresponds to the condition m exists in P in the semantics. The value of `exists` is computed once and for all in the constructor. When statically matching a join point, `exists` is tested first (line 21). If it is false, the selector directly returns false. Another possibility would be not to capture the `NoMethodException` in the constructor but rather throw an exception at the level of the constructor. In that case[2], it would not be possible to create a call selector for which the method would not exist in the pointcut declaring type: the boolean variable `exists` is not necessary.

- The boolean variable `basicMatch` is set to true (line 29) if the join point is indeed a method-call join point (line 23) for the method specified in the pointcut (lines 24 and 25). This requires to compare the name and parameter types specified in the pointcut and the ones obtained from the join point (line 25).

If both `exists` and `basicMatch` are true, the last step consists of checking (line 28), using the method `asSubClass` of the Java reflection API, that the qualifying type of the join point,

[2]When using the AspectJ Eclipse plugin, a warning is emitted when the method does not exist, but this is not considered as an error.

```java
public class Call extends JoinPointSelector {
  public Class pointcutClass;
  public String methodName;
  public Class[] parameterTypes;
  public boolean exists;

  public Call(Class pointcutClass, String methodName,
              Class[] parameterTypes)
      throws NullPointerException, SecurityException {

    try {
      pointcutClass.getMethod(methodName, parameterTypes);
      exists = true;
    } catch (NoSuchMethodException) {
      exists = false;
    }
  }

  public boolean staticTest(JoinPoint jp) {
    boolean basicMatch;
    if (exists) {
      basicMatch =
        jp.getKind().equals("method-call")
        && jp.getSignature().getName().equals(methodName)
        && Arrays.equals(((MethodSignature)jp.getSignature()).
            getParameterTypes(), parameterTypes);
      if (basicMatch) {
        try {
          jp.getSignature().getDeclaringType().asSubclass(
              pointcutClass);
          return true;
        } catch (Exception e) {
          return false;
        }
      } else
        return false
    } else
      return false;
  }
  public boolean dynamicTest(JoinPoint jp) {
    return true;
  }
}
```

Listing 6 – *The selector* `Call`.

```
public class Execution extends JoinPointSelector {
    ...
    basicMatch = jp.getKind().equals("method−execution");
    ...
}
```

Listing 7 – *The selector* `Execution`*.*

```
public class Target extends JoinPointSelector {
  Class pointcutClass;

  public Target(Class pointcutClass) {
    this.pointcutClass = pointcutClass;
  }

  public boolean staticTest(JoinPoint jp) {
    return true;
  }
  public boolean dynamicTest(JoinPoint jp) {
    return jp.getTarget().getClass().equals(pointcutClass);
  }
}
```

Listing 8 – *The selector* `Target`*.*

`jp.getSignature().getDeclaringType()`[3], is a subclass of the pointcut declaring class `pointcutClass`. This directly corresponds to the condition $J <: P$ in the semantics.

4.3 Execution

The implementation of the execution selector is exactly the same as the implementation of the call selector with the difference that we test if the join point kind is "method−execution". This similarity directly follows from the fact that the semantics of the corresponding pointcuts are exactly the same, modulo the definition of the join-point qualifying types, and from the fact that we use AspectJ join points. When calling `getSignature.getDeclaringType()`, we get the static type of the receiver object for method calls and the type which defines the executed method for method executions.

4.4 Target

Listing 8 shows the implementation of the target selector. It is very simple because we use the capabilities provided by the AspectJ join point API to access the target object and to compare its type with the pointcut declaring class. Note that the pointcut `target`, with the current semantics of AspectJ, deals only with dynamic information (target type). For this reason, the method `staticTest` always returns `true`.

[3]A more consistent name would be `getQualifyingName()`.

```
public boolean staticTest(JoinPoint jp) {
  boolean basicMatch;
  if (exists) {
    ...
    return basicMatch;
  } else
    return false;
}
public boolean dynamicTest(JoinPoint jp) {
  try {
    jp.getTarget().getClass().asSubclass(pointcutClass);
    return true;
  } catch (Exception e) {
    return false;
  }
}
```

Listing 9 – *The* dynamic *selector* `Call`.

4.5 Alternative Semantics

In the following, we show the minor changes that are necessary to the previous implementation in order to get the dynamic semantics of Barzilay et al. (2004).

4.5.1 Call with Dynamic Semantics

The conditions are the same as with the static semantics except that we need to replace the qualifying type of the join point given by AspectJ by its dynamic type, that is `jp.getSignature().getDeclaringType()` by `jp.getTarget().getClass()`. We use again the facilities provided by the AspectJ join points API.

Also, as the dynamic type is, as its name indicates, dynamic, the related computations should be moved from the method `staticTest` to the method `dynamicTest`.

Starting from Listing 6 and applying these changes, we get Listing 9. The true branch of the conditional `if(basicMatch)` of the method `staticTest` is moved to the method `dynamicTest` while now testing the dynamic type of the join point rather than its AspectJ qualifying type. In the method `staticTest`, the conditional is replaced by a simple `return (basicMatch)` statement.

4.5.2 Execution with Dynamic Semantics

The principles are the same as before. The differences are that, when computing `basicMatch` "method−call" should be replaced by "method−execution", and that the dynamic type of the join point is accessed through `jp.getThis()` instead of `jp.getTarget()`.

5 Conclusion

In this paper, we have proposed an interpreted approach that facilitates the implementation of various pointcut semantics. It is based on a representation of a pointcut as an instance of the class `JoinPointSelector` with two methods, `boolean staticTest(JoinPoint)` and `boolean dynamicTest(JoinPoint)`, dealing with static and dynamic information, respectively. It also uses a two-step weaving scheme, which makes it possible to use AspectJ to instrument the base program and benefits from its basic facilities to access join point information. Of course, a basic limitation of the approach is that only AspectJ join points can be taken into account, unless the AspectJ layer is, for instance, extended with other instrumentation means.

As a proof of the versatility of this approach, we have shown how to implement the current semantics of the pointcuts `call` and `execution` as well as the alternative dynamic semantics of (Barzilay et al., 2004). These implementations can be deduced very straightforwardly from their semantics. The versatility of the approach has however a cost resulting from the introduction of an interpretation layer. We think that, depending on the application, the overhead may remain reasonable if the instrumentation performed by the aspect `Platform` is tailored in order to limit the instrumentation to the shadows of interest, but this remains to be quantified. On the same vein, one can imagine to mix the AspectJ semantics and alternative semantics, for instance by introducing new pointcuts with alternative semantics (this could apply to other features as well), and limit the use of interpretation to non-standard semantics. One other interesting perspective allowed by the versatility of the approach is to refine the implementation in order to execute AspectJ applications with the AspectJ semantics but check whether the result would have been different using an alternative semantics.

We have presented the implementation of the pointcuts `call` and `execution` but this approach can be used to define any other pointcut including new types of pointcuts, for instance domain-specific pointcuts.

This work is actually part of a larger framework that we call CALI (Assaf, 2008; Assaf and Noyé, 2008), for *Common Aspect Language Interpreter*. CALI is used to prototype aspect languages and study multi-aspect language AOP. CALI as well as examples of AOPL prototypes can be downloaded from (Assaf, 2008).

Acknowledgments

We are thankful to the reviewers for their very helpful comments.

References

Assaf, A. (2008). CALI homepage. http://www.emn.fr/x-info/cali.

Assaf, A. and J. Noyé (2008). Dynamic AspectJ. In J. Brichau (Ed.), *DLS '08: Proceedings of the 2008 symposium on Dynamic languages*, Paphos, Cyprus, pp. 1–12. ACM.

Avgustinov, P., A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble (2005). abc: an extensible AspectJ compiler. In

M. Mezini and P. L. Tarr (Eds.), *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, Chicago, Illinois, USA, pp. 87–98. ACM.

Avgustinov, P., E. Hajiyev, N. Ongkingco, O. de Moor, D. Sereni, J. Tibble, and M. Verbaere (2007). Semantics of static pointcuts in AspectJ. In M. Hofmann and M. Felleisen (Eds.), *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2007)*, Nice, France, pp. 11–23. ACM.

Barzilay, O., Y. A. Feldman, S. Tyszberowicz, and A. Yehudai (2004). Call and execution semantics in AspectJ. In G. T. Leavens, C. Clifton, and R. Lämmel (Eds.), *FOAL 2004 Proceedings - Foundations of Aspect-Oriented Languages - Workshop at AOSD 2004*. Department of Computer Science, Iowa State University.

David, P.-C., T. Ledoux, and N. M. Bouraqadi-Saâdani (2001). Two-step weaving with reflection using AspectJ. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*.

Djoko Djoko, S., R. Douence, P. Fradet, and D. Le Botlan (2006). CASB: Common Aspect Semantics Base. Deliverable D54, AOSD-Europe.

Filman, R. E., T. Elrad, S. Clarke, and M. Akşit (Eds.) (2005). *Aspect-Oriented Software Development*. Boston, USA: Addison-Wesley.

Hilsdale, E. and J. Hugunin (2004). Advice weaving in AspectJ. In K. Lieberherr (Ed.), *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, Lancaster, UK, pp. 26–35. ACM.

Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold (2001). An overview of AspectJ. In J. L. Knudsen (Ed.), *ECOOP 2001 - Object-Oriented Programming, 15th European Conference*, Number 2072 in Lecture Notes in Computer Science, Budapest, Hungary, pp. 327–353. Springer-Verlag.

Kojarski, S. and D. H. Lorenz (2006). Modeling aspect mechanisms: a top-down approach. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa (Eds.), *28th International Conference on Software Engineering (ICSE 2006)*, Shanghai, China, pp. 212–221. ACM.

The AspectJ Team (2003). *The AspectJ Programming Guide*. Xerox Corporation and Palo Alto Research Center.

Résumé

Un des éléments principaux d'un langage ou d'un canevas de programmation par aspects est son langage de coupe. Une coupe est un prédicat qui sélectionne les points d'exécution qui doivent être affectés par un aspect. L'utilisation d'AspectJ montre que les deux coupes primitives de base, `call` et `execution`, qui traitent respectivement de l'appel et de l'exécution d'une méthode, peuvent être source de confusion du fait de subtiles interactions entre l'utilisation de types statiques ou dynamiques pour sélectionner les points d'exécution, de la recherche dynamique des méthodes et de l'attente d'une sélection facile des points d'exécution côté appelant et appelé d'une même invocation. Aussi, des sémantiques alternatives, plus intuitives, ont été proposées mais en sont restées à l'état de la conception.

Dans cet article, nous reconsidérons un certain nombre de sémantiques possibles d'une manière pratique, en les implémentant à l'aide de CALI, notre interprète général de langages d'aspects. Ce canevas utilise Java comme langage de base et AspectJ comme moyen d'extraire les points d'exécution à considérer. Une couche d'interprétation additionnelle peut alors être utilisée pour prototyper d'intéressantes variantes de programmation par aspects dans un environnement complet. L'article utilise les coupes `call` et `execution` pour illustrer les bénéfices d'un tel dispositif. Il montre que des alternatives peuvent être implémentées très facilement et expérimentées dans le contexte d'AspectJ sans avoir recours à une technologie de compilation rapidement complexe.