Predicate dispatch for Aspect-Oriented Programming

(Position paper)

Shigeru Chiba

Tokyo Institute of Technology
2-12-1 Ohkayama, Meguro-ku,
Tokyo 152-8552, Japan
www.csg.is.titech.ac.jp/~chiba

Abstract

Developing a machine model natively supporting aspect-oriented programming (AOP) is fruitful not only for implementing interpreters and compilers for AOP languages but also for understanding the essence of the AOP paradigm. This position paper shows a machine model based on open classes and predicate dispatch and it briefly describes a list of predicates that are necessary for modeling AspectJ. This machine model is useful for comparing AOP and object-oriented programming (OOP) on a side-by-side basis. Our initial observation is that AOP is a natural extension to OOP with respect to language constructs.

1. Introduction

Early aspect-oriented programming (AOP) languages such as AspectJ were implemented as a program translator into a program (or machine code) written in a non-AOP language. These days, several virtual machines that directly support AOP such as Steamloom [2] have been developed but their support is still limited [5]. This is partly because of the lack of a good machine model with native support for AOP.

To develop such a machine model for AOP, Haupt and Schippers proposed the concept of virtual join points [5] and showed the delegation-based AOP model based on that concept. In their paper, they presented similarity between join points in AOP and method calls in object-oriented programming (OOP) and AOP makes join points *virtual* as OOP makes function calls virtual (if we follow the C++ terminology). Here, join points are execution points in which two pieces of code are connected statically or slightly dynamically by the dynamic method dispatch of OOP. AOP gives another kind of late-binding feature to the join points. The delegation-based AOP model implements this virtualization by inserting a proxy object into a message delegation chain. In this model, all objects are prototype based and a method dispatching mechanism is represented by message delegation. Although a proxy can implement various kinds of advice, this model requires the delegation-chain mechanism to be extended for supporting a new pointcut primitive. For example, as shown in their paper, to support the cflow pointcut, they had to introduce a new kind of delegation chain that is effective only for a particular thread. Since the delegation chain is a fundamental component of the prototype-based object system, it should not be modified to support a new pointcut primitive.

This paper presents another machine model for AOP. It is still based on the concept of virtual join points but it uses an extended version of predicate dispatch [4, 6] as the basic mechanism. We believe that predicate dispatch is more intuitive for implementing the virtual join points than the delegation chain. Presenting similarity between predicate dispatch and the pointcut-advice of AOP is not new. This fact has been pointed out in a few papers [7, 1, 5]. The contribution of this paper is to show what predicates are really necessary for emulating pointcuts provided by AspectJ. We believe that our model is also useful to understand AOP by side-by-side comparison with OOP because predicate dispatch is a natural extension to the dynamic method dispatch of OOP. It would be also possible to introduce good ideas, such as the ambiguity property, for method dispatching in OOP into AOP.

2. Basic constructs

In our model, objects are instances of classes, which can consist of multiple modules as in open classes [3] and Hyper/J [8]. Suppose that we have the following class:

```
class Shape {
  Position p;
  Position getPos() { return p; }
  void setPos(Position np) { p = np; }
```


VMIL'08, October 19, 2008, Nashville, TN.

```
}
```

Then we can define a complementary module that adds some members to the Shape class. It would look something like this:

```
class Updater refines Shape {
  Display d;
  void setDisplay(Display nd) { d = nd; }
  void setPos(Position np) {
    d.repaint();
    // call a less-specific method with np.
  }
}
```

This Updater class corresponds to an aspect in the sense of AspectJ. We can define any number of classes that refine the Shape class. Updater itself is not a complete class; it cannot be instantiated. It is the name of a part of the declaration of the Shape class.

The Updater class adds a field d and a method setDisplay. These correspond to intertype declaration. Thus, at runtime, the Shape class has not only p, getPos, and setPos but also the added members d and setDisplay. An instance of the original Shape class includes those added members even if it is created by the code that does not expect the existence of Updater at all.

Note that the Updater also adds a method setPos, which is also declared in Shape. It is an advice. In this model, advising a join point is represented by overriding a method. The setPos method in Updater overrides the setPos method in Shape as an around advice in AspectJ does. Although we use the term *override*, the semantics of our overriding is slightly different from the normal one. The two implementations of setPos method given by Updater and Shape belong to the Shape class at runtime. However, since the implementation given by Updater *overrides* the other, it is invoked when the setPos method is called on an instance of Shape. No "multiple methods are applicable" error will be thrown. This is the same as the behavior of an around advice with the execution pointcut, which is invoked instead of the method specified by that execution pointcut when it is called.

3. Predicate dispatch

Since AspectJ provides several pointcut designators, simple method overriding is not sufficient to model AspectJ. Thus, we introduce an extended version of predicate dispatch. In a language supporting predicate dispatch, such as JPred [6], a method declaration can include predicates and it is invoked only when all the predicates are true. The original predicate dispatch allows only predicates that take local contexts such as parameter values and the receiver object (*i.e.* the this variable in Java). For example, JPred provides a predicate that becomes true if a parameter value is an instance of a specific class. This restriction is necessary for modular type checking and compilation.

However, since AOP is a paradigm for dealing with a crosscutting concern, our model for AOP needs some predicates that deal with non-local contexts. In other words, enabling method dispatch depending on non-local contexts is a unique feature of AOP against OOP. For example, see the following aspect in AspectJ:

```
aspect Logging {
  around(): call(void HashMap.put(..))
            && within(WebApp)) {
    System.out.println(
            "WebApp updates a hash map");
    proceed();
  }
}
```

This typical logging aspect declares an advice that is invoked when any method in the WebApp class calls the put method on a HashMap object. If a method in other classes calls it, the advice is not invoked but the original put method is invoked.

In our model, this advice is interpreted as a method overriding the put method in the HashMap class only when the client object that calls put is an instance of WebApp. Hence we need a predicate that checks the type of the client object, which is part of non-local contexts. The aspect described by using our model would be something like this:

```
class Logging refines HashMap {
  void put(Object key, Object value)
          when client instanceof WebApp {
    System.out.println(
            "WebApp updates a hash map");
    proceed();
  }
}
```

The expression following when is the predicate for the put method. Here, client is a hidden parameter, which is available without explicit declaration. It refers to the client object that calls the method.

Our model accepts a method with some predicates. The method with predicates does not have to override another method that has no predicates. For example, a method m may have only the implementations with predicates; in certain runtime contexts, all the implementations for m might be ineffective and thus a "message not understood" error would occur because all their predicates are false. However, we can modify our model to fit AspectJ's semantics. In AspectJ, an advice always modifies the behavior of an existing method. To implement this semantics in our model, we must constrain all methods with predicates to override the *default* method, which has the same name and signature but no predicates.

The predicates available in our model is defined as the following:

<*predicates*> := <*predicate*>

| <*predicate*> <*op*> <*predicates*>

<*predicate*> := <*var*> instanceof <*type*>

| <*var*> statically-instanceof <*type*>

| <*var*> running <*method*>

| ! <*predicate*>

| (<*predicate*>)

<*var*> := this | client | client* | <*parameter*>

<*op*> := && | ||

As <*var*>, not only this (*i.e.* a callee object) and a call parameter but also client is available to indicate the client object that calls the method. Thus, instanceof predicates can represent the same conditions that this, target, args, and within pointcuts in AspectJ can do. statically-instanceof is for the call pointcut in AspectJ. It checks the static type of <*var*>, usually this. If the static type of the receiver object is <*type*> at the client site, this predicate:

this statically-instanceof <*type*>

becomes true. The static types of client and the parameters are always the same as their dynamic types. running is for the withincode and cflowbelow pointcuts. It becomes true if <*var*>, which is usually client or client*, is running the specified <*method*>. client* represents any object contained in the current call stack. Hence,

client* running <*method*>

becomes true only while the current thread of control is executing the specified <method>. This corresponds to the cflowbelow pointcut.

3.1 Exhaustiveness

The original predicate dispatch was carefully designed for the exhaustiveness property. This property guarantees that no "message not understood" error happens during runtime. For example, the compiler of JPred [6] can modularly check that this property is preserved. If predicate dispatch is naively available, it is not obvious whether or not a program has the exhaustiveness property because some methods may have implementations effective only under certain runtime conditions. If the compiler allows calling such a method, the call would cause a "message not understood" (*i.e.* the called method is not found) error when there is no implementation effective for the calling contexts.

Unfortunately, our model does not enable static exhaustiveness checks because a predicate may access client*, which is never statically determined. On the other hand, our modified model allows a compiler to statically check that a given program preserves the exhaustiveness property. Recall that we mentioned that, to make our model exactly fit AspectJ's semantics, we have to add an extra constraint to our model. Since this modified model constrains a method with predicates to override the default method, which does not have predicates, any method has an implementation that is always effective independently of runtime contexts. Thus, the compiler can statically check exhaustiveness if all the classes including *aspect* classes refining normal classes are known to the compiler. Since our model adopts open classes, the set of the methods available in a given class is not determined unless all the modules contributing to that class (a normal class and aspect classes refining it) are given.

To enable modular exhaustiveness checks, we must introduce another language construct. For example, as eJava [9] does, the client code must explicitly specify modules that declare methods used by that client code. If the client code uses the setDisplay method added by the aspect class view.Updater to the original class model.Shape, it must include the following statement:

use view.Updater;

Then the compiler can understand that the setDisplay method in the model.Shape class is declared in the source file of the view.Updater class, which refines model.Shape. This enables modular class-by-class compilation as the Java compiler does.

Note that the explicit import of modules via use statement is not necessary when client code invokes an overriding method given by an aspect class. The compiler can perform modular exhaustiveness checks if the called method is also declared in the original class or another aspect class explicitly imported via use. Suppose that the refining class view.Updater overrides the setPos method in the model.Shape class. The compiler can modularly perform exhaustiveness check on the client code that will call the setPos method whichever implementation will be invoked, view.Updater's or model.Shape's.

3.2 Ambiguity

Another interesting property is ambiguity. If a program is not ambiguous, a "multiple methods are applicable" error never occurs during runtime. Ambiguity checking ensures that there are no two methods that have the same name and signature and also have predicates that become true in the same runtime contexts.

Since AspectJ allows multiple advices modifying the behavior of the same join point, our model does not guarantee that all programs preserve the ambiguity property. Two aspect classes refining the same normal class may override the same method and their two overriding methods might be effective at the same time.

In AspectJ, programmers can explicitly specify the precedence order among aspects. We can introduce a similar mechanism into our model so that the ambiguity of method dispatch will be resolved. Then a compiler will be able to conservatively check that all necessary precedence order is given.

We use predicates for representing the precedence order among aspect classes. The most specific method among ef-

fective overriding methods is determined by using logical implication relations among predicates. Note that the original predicate dispatch also uses implication relations for determining the most specific method. If two methods m_1 and m_2 are included in the effective methods for a method call and m_1's predicate expression logically implies m_2's predicate expression, then m_1 is more specific method than m_2. For example, suppose that m_1's predicate expression is p1 instanceof Rect and m_2's predicate expression is p1 instanceof Shape, where p1 is the first parameter of m_1 and m_2 and Rect is a subclass of Shape. Then m_1 is more specific than m_2 because, if p1 is an instance of Rect, then it is always an instance of Shape.

We implement the precedence order by specializing this algorithm for determining the most specific method. Let us introduce a new predicate deploy. It takes one aspect class name as a parameter. For example, deploy(Logging) is true only when an aspect class named Logging is deployed. If another aspect class Updater is also deployed and Logging has higher precedence than Logging, then deploy(Logging) logically *implies* deploy(Updater). The parameter of deploy can be null. deploy(null) is logically implied by deploy(A) for any aspect class A. In summary,

deploy(*A*) logically implies deploy(null) for any *A*.

deploy(*A*) logically implies deploy(*B*)
if *A* has higher precedence than *B*.

In our model, any method implicitly has a deploy predicate. If the method is declared in an aspect class A , which refines another class, then it has deploy(A). Otherwise, if it is declared in a normal class, it has deploy(null). If two methods m_1 and m_2 are effective and m_1's deploy logically implies m_2's deploy, then m_1 is a more specific method than m_2 and hence m_1 overrides m_2. Other predicates are not used to determine which method is more specific although it is possible to enhance our model to use other predicates.

3.3 super **and** proceed

By introducing a deploy predicate, we could make our model show similar behavior as AspectJ. However, to make our model exactly emulate AspectJ's semantics, we need more.

AspectJ has an algorithm for determining which advice is first executed, that is, which method is the most specific. However, this algorithm is not modular. Suppose that a class Shape has a method setX. An aspect Logging declares an advice α for calls to setX. There is another class Rect, which is a subclass of Shape and overrides the method setX. Calls to the method setX in Rect is advised by an advice β declared in an aspect Updater. The aspect Logging has higher precedence than Updater.

If the setX method is called on an instance of Rect, then the advice α is first executed because Logging has the highest precedence. If α calls proceed, then the advice β is next executed. After that, the method setX in Rect is executed. If setX calls super.setX, then the advice α is executed again, and finally the setX in Shape is executed. The order of specificity for each method is α, β, Rect.setX, α (again), and Shape.setX. Note that the advice α is executed twice. Although α advises the setX method in the super class, it is executed for the call to the method in the subclass and it is executed again for the call on super.

To emulate this behavior, our model also has to support both super and proceed to invoke a less specific method:

- proceed(p1, p2, ...)
  This invokes a less specific method with parameters p1, p2, ... The order of specificity is determined by deploy predicates and declaring classes. This is usually used by methods in aspect classes.
- super.<*method*>(p1, p2, ...)
  This calls the same method on the same instance with using the super class as the dynamic type of that instance. This is usually used by methods in normal classes.

For the example above, the setX method in Rect should call super.setX(newX) in our model. The setX methods in the aspect classes Logging and Updating should call proceed(newX). Then the behavior of a call to setX on an instance of Rect is the same as in AspectJ.

4. Concluding remarks

This paper presents an AOP machine model based on open classes and predicate dispatch. This model leads us to regard AOP as a natural extension to OOP. The paper argued that the language constructs of AOP are fairly equivalent to an extended version of predicate dispatch and thereby it is possible to directly compare AOP with other OOP-based programming paradigms. The contribution of this paper is to show what predicates are really necessary for emulating pointcuts provided by AspectJ. The original predicate dispatch allows only predicates that take local contexts such as parameter values and the receiver object. This restriction is for modular type checking and compilation. On the other hand, to emulate pointcuts, we had to allow predicates that deal with non-local contexts such as the client object. This is necessary for modularizing crosscutting concerns. A drawback of this fact is that modular type checking and compilation is sacrificed.

The presented work is still at a very early stage and this paper shows only a very rough sketch of the machine model. A lot of work remains. Our model does not support pattern matching or several pointcuts such as if, handler, set, or get. It does not support aspect instances.

References

[1] Bockisch, C., M. Haupt, and M. Mezini, "Dynamic Virtual Join Point Dispatch." Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT '06), 2006.

[2] Bockisch, C., M. Haupt, M. Mezini, and K. Ostermann, "Virtual machine support for dynamic join points," in *Proc. of Int'l Conf. on Aspect-Oriented Software Development (AOSD 2004)*, pp. 83–92, 2004.

[3] Clifton, C., G. T. Leavens, C. Chambers, and T. Millstein, "MultiJava: modular open classes and symmetric multiple dispatch for Java," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 130–145, ACM Press, 2000.

[4] Ernst, M., C. Kaplan, and C. Chambers, "Predicate Dispatching: A Unified Theory of Dispatch," in *ECCOP '98: Proc. of the 12th European Conference on Object-Oriented Programming*, pp. 186–211, Springer-Verlag, 1998.

[5] Haupt, M. and H. Schippers, "A Machine Model for Aspect-Oriented Programming," in *ECOOP 2007 – Object-Oriented Programming*, vol. 4609 of *LNCS*, pp. 501–524, 2007.

[6] Millstein, T., "Practical predicate dispatch," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 345–364, ACM, 2004.

[7] Orleans, D., "Separating behavioral concerns with predicate dispatch, or, if statement considered harmful," in *Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA '01*, 2001.

[8] Ossher, H. and P. Tarr, "Hyper/J: multi-dimensional separation of concerns for Java," in *Proc. of the Int'l Conf. on Software Engineering (ICSE)*, pp. 734–737, 2000.

[9] Warth, A., M. Stanojević, and T. Millstein, "Statically scoped object adaptation with expanders," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 37–56, 2006.