AspectScope: An Outline Viewer for AspectJ Programs

Michihiro Horie, Tokyo Institute of Technology, Japan
**Shigeru Chiba**, Tokyo Institute of Technology, Japan

This paper presents the *AspectScope* tool for AspectJ. It displays how aspects affect the existing module interfaces in the program. Because an aspect is not explicitly invoked, some developers claim that it is difficult to understand the behavior of their code within local reasoning. Although this difficulty should be mitigated by appropriate tool support, the support by current tools such as AJDT is not sufficient. We have developed AspectScope for providing another visualization of AspectJ programs so that developers can more easily understand crosscutting structures in their programs.

1   INTRODUCTION

The pointcut and advice mechanism of Aspect-Oriented Programming (AOP) languages such as AspectJ [9] allows developers to combine a module to a special module, called an aspect, without explicit method calls. This is useful to implement certain crosscutting concerns as a separate module. An aspect is implicitly invoked when a thread of control reaches some execution points in the other module. Those execution points are selected from the predefined set of points by the language.

However, this property of AOP makes it difficult for developers to understand the behavior of a module as long as they are looking at only the source code of that module. When one module is executed in an AOP language, other modules might be implicitly invoked from that module. The behavior might be changed by the deployment of other modules (*i.e.* aspects). Therefore, AOP languages require a whole-program analysis for understanding a program.

To address this problem, several programming tools for AOP have been developed. One of the most popular tools is AJDT, AspectJ Development Tools of Eclipse IDE [20]. It automatically performs a whole-program analysis and visualizes the crosscutting structures in the program according to the result of the analysis. The developers do not have to manually perform a whole-program analysis any more. However, AJDT does not seem to satisfy developers. Their claim is that they want to see *static* module interfaces for understanding their programs. Here, the module interfaces include the specifications of the behavior of the modules. Although AJDT automatically performs a whole-program analysis while a developer is editing a program, the visualization by AJDT does not much help the developer see the module interfaces. It does nothing except simply showing the join points where modules are combined with aspects. Even worse, module interfaces in AOP languages are never
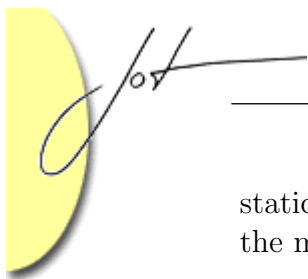
static or stable. It changes according to the deployment of aspects. In this sense, the module interfaces in AOP are essentially different from traditional ones.

Although module interfaces in AOP are hard to see, it should be possible to improve the visualization by a programming tool so that developers can more easily see the module interfaces under the current deployment of aspects. This would hopefully give better impression of AOP to the developers, who want to reason about their programs at a module level.

This paper presents *AspectScope*, which is our programming tool for AspectJ. We have developed it for realizing our idea above. Like AJDT, it automatically performs a whole-program analysis and visualizes the result. However, it shows how aspects affect module interfaces in the program. It interprets an aspect as an extension to other classes and it displays the extended module interfaces of the classes under the deployment of the aspects. It thereby helps developers understand crosscutting structures in the program.

In the rest of this paper, Section 2 mentions the limitations of AJDT with respect to the visualization of crosscutting structures. Section 3 presents AspectScope, which is our programming tool. Section 4 presents examples of the use of AspectScope. Section 5 describes related work and Section 6 concludes this paper.

2 MODULAR REASONING

The standard AspectJ support of Eclipse IDE, named AJDT [20], visualizes a crosscutting structure in an AspectJ program. This helps developers to reason about the program with a modular fashion despite the obliviousness property of AspectJ [7]. However, the help by this visualization is still limited and thus developers sometime feel that AOP makes modular reasoning difficult.

To illustrate the limitation of AJDT, we below show a refactoring process of a figure editor [10] as an example scenario. A figure editor is a simple tool for editing drawings that are composed of points and lines. Since a display of the tool must always reflect the current states of such shapes, any method that is declared in Point or Line class must call the update method in the Display class whenever that method changes the states of shapes. The update method redraws a display so that the pictures of all the shapes such as points and lines on the display will be updated. Figure 1 shows the AspectJ program of this figure editor. The concern of updating a display is implemented in an aspect.

We then perform simple refactoring. The x and y field in the Point class are intentionally public. Since this fact is obviously a weakness in information hiding, suppose that a developer changes these fields to being private. This change causes another change in the moveBy method in the Line class. The fields x and y on p1 and p2 are not accessible any more. The developer must change the moveBy method. Figure 2 shows the new revision of moveBy method.

```
public interface Shape {
  void moveBy(int dx, int dy);
}

class Line implements Shape {
   private Point p1, p2;
   public void setP1(Point np1) {p1 = np1;}
   public void setP2(Point np2) {p2 = np2;}
   public Point getP1() {return p1;}
   public Point getP2() {return p2;}
   public void moveBy(int dx, int dy) {
      p1.x += dx; p1.y += dy;
      p2.x += dx; p2.y += dy;
   }
}
```

```
class Point implements Shape {
  public int x, y;   // intentionally
  public void setX(int nx) {x = nx;}
  public void setY(int ny) {y = ny;}
  public int getX() {return x;}
  public int getY() {return y;}
  public void moveBy(int dx, int dy) {
     x += dx; y += dy;
  }
}

aspect UpdateSignaling {
  pointcut change():
     call(void Point.setX(int))
     || call(void Point.setY(int))
     || call(void Shape+.moveBy(int,int));

  after() returning: change() {
     Display.update();
  }
}
```

Figure 1: A figure editor implemented in AspectJ
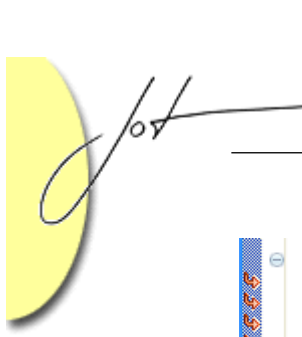
```
public void moveBy(int dx, int dy) {
   p1.setX(p1.getX() + dx);
   p1.setY(p1.getY() + dy);
   p2.setX(p2.getX() + dx);
   p2.setY(p2.getY() + dy);
}
```

Figure 2: The new revision of moveBy method

Unfortunately, the refactoring has not finished yet. If the moveBy method is invoked, the developer will see that the display flickers. To understand this problem, the developer will have to investigate the whole program including aspects and find which join points are advised. Local investigation within the moveBy method or the Line class does not reveal the problem to the developer because of the obliviousness property of AspectJ. The lexical representation of the moveBy method does not contain any sign or symptom of being advised.

AJDT helps the investigation. It automatically performs a whole-program analysis and visualizes which join points are advised by an aspect. See four arrow icons at the left side of the source editor in Figure 3. The developer can notice that the four calls to setX and setY in the body of moveBy are advised by the UpdateSignaling aspect, which invokes the update method in the Display class to repaint the display and cause a flicker.
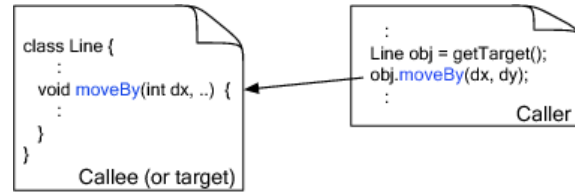
Figure 3: AJDT indicates advised join points

Figure 4: The caller side and the callee side

However, the help by AJDT is not sufficient. The developer, who saw Figure 3, would change the moveBy method so that only the first call to setX would be advised. Suppose that she changes the moveBy method to the following:

```
public void moveBy(int dx, int dy) {
  p1.setX(p1.getX() + dx);
  p1.incY(dy);
  p2.incX(dx); // incX() and incY() are not advised
  p2.incY(dy);
}
```

She also adds new methods incX and incY for incrementing the value of x or y. We assume that calls to these new methods are not advised. Unfortunately, this change does not stop a flicker. Although now only one join point (*i.e.* a call to setX) in the body of the moveBy is advised, a call to the moveBy method itself is also advised. Therefore, each call to the moveBy method causes two successive invocations of the update method and they cause a flicker.

The problem here is that the developer cannot notice that a call to moveBy is also advised as long as she is looking at the source code of the moveBy method. AJDT does not tell her the fact unless she opens a client class of Line and then looks at a *caller-side* method of moveBy (Figure 4). If a pointcut is call, AJDT puts an arrow icon only at a method-call expression that calls the advised method. It does not show any indications at the *callee-side*. Note that a call pointcut selects join points at which method-calls are executed in a client class, while an execution pointcut selects join points at which method bodies are executed. Thus, to reach a right solution, the developer must manually perform a whole-program analysis to a certain degree and understand the crosscutting structure in the program. Then she must edit the aspect program so that the update method will be invoked only once for each top-level change of the state of the shape. The revised UpdateSignaling aspect is the following:

```
after() returning:  change() && !cflowbelow(change()) {
  Display.update();
}
```
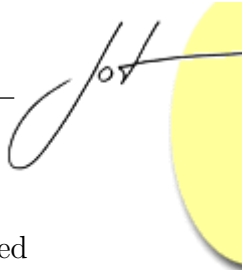
Now the **update** method is invoked only when either **setX**, **setY**, or **moveBy** is called as a top-level call. Since a **cflowbelow** pointcut selects join points below the control flow of the specified join points, **update** is invoked only once for each call to the **moveBy** method. The developer does not have to add **incX** or **incY** to the **Point** class.

3 ASPECTSCOPE

Although AJDT visualizes crosscutting structures in a program, it only indicates where a crosscutting structure joins other structures, that is, it only indicates join points in the source code. As we have seen in the previous section, this visualization is not sufficient to help developers understand crosscutting structures in their programs.

For better help, we have developed another programming tool for AspectJ. It is an Eclipse plugin named *AspectScope*. This tool visualizes crosscutting structures by showing how aspects affect the module interfaces in the program. Like AJDT, the tool performs a global analysis of the deployment configuration of aspects but it presents the result of the analysis from the viewpoint of how the module interfaces of classes are extended by aspects. In other words, our tool projects AOP structure onto normal OOP (Object-Oriented Programming) structure so that developers can see crosscutting structures through their familiar OOP view. For example, the tool does not distinguish the **call** pointcut and the **execution** pointcut because the influence of these pointcuts on module interfaces is equivalent. It abstracts away from language-level differences between **call** and **execution**.

AspectScope consists of two panes: one for showing an outline view of a given class and the other for presenting javadoc comments describing the behavior of a selected method or field. These two panes reflect the extensions by woven aspects. See Figure 5.

Outline view

The outline view by AspectScope lists methods and fields declared in a given class. It also shows whether or not the behavior of each method or field is extended by an aspect.

The execution and call pointcuts:

If an **UpdateSignaling** aspect includes an **after** advice associated with a pointcut **execution(void Point.setX(int))**, then the outline view indicates that the **setX** method in the **Point** class is extended by the **after** advice in the **UpdateSignaling** aspect (Figure 6).
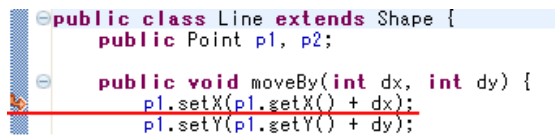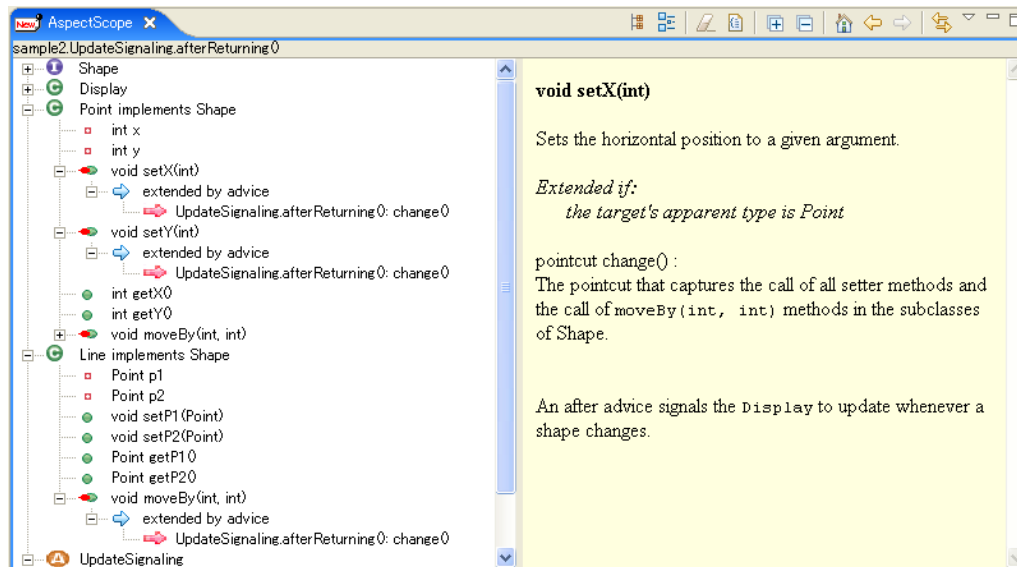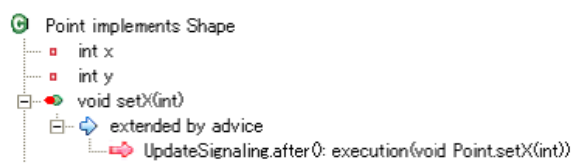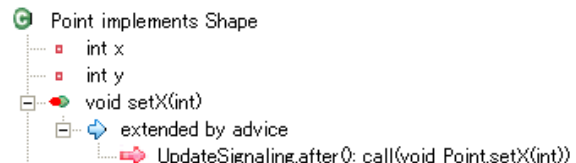
Figure 5: AspectScope

Figure 6: The outline view presents the effect of the execution pointcut.

Figure 7: The outline view presents the effect of the call pointcut.

Note that even if the pointcut that the after advice is associated with is not execution but call, for example, call(void Point.setX(int)), then the outline view shown does not change except the description of the pointcut (Figure 7). AspectScope abstracts away from differences between call and execution because module interfaces affected by aspects are interesting concerns. AspectScope considers that the advice associated with either pointcut extends the behavior of the *callee-side* method. In AspectJ, both pointcuts select method calls. However, the join points (or join point shadow [12]) selected by a call pointcut are method-call expressions at the *caller* side while the join points selected by an execution pointcut are the bodies of the specified methods at the *callee* (or *target*) side. Hence, for example, the advice associated with a call pointcut can obtain a reference to not only the target object but also the caller object. On the other hand, the advice associated with an execution pointcut cannot obtain such a reference.

Despite this difference, AspectScope uses the outline view of the *callee* side to indicate the extension by the call pointcut. Since the goal is to display the module interfaces affected by aspects, AspectScope must project the extension to a module interface, which is the outline view of the callee side in OOP. On the other hand,

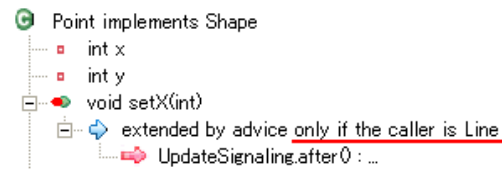Figure 8: AJDT indicates the effect of the call pointcut (the red underline was drawn by the authors).

Figure 9: A conditional extension by the within pointcut (the red underline was drawn by the authors)

AJDT reflects this difference. Figure 8 illustrates AJDT's visualization of the call pointcut shown above. An arrow icon indicates that the call to setX within the moveBy method is one of the selected join points. Note that the source code in this figure is of the caller-side method moveBy. AJDT does not show any information in the source code of the setX method, which is at the callee side.

The within and cflow pointcuts:

The within, withincode, cflow, and cflowbelow pointcuts select join points within a specified region. For example, the within pointcut selects only the join points included in the specified class. call(void *.setX(int)) && within(Line) selects method calls from the Line class to setX declared in any class. The selected join points are method-call expressions contained in the body of a method in the Line class. The within pointcut restricts the *caller* methods.

If the call pointcut is combined with the within pointcut, AspectScope interprets that the associated advice conditionally extends the behavior of the *callee* method. This is also true for the combination of call and cflow, set and within, and so forth. For example, if an UpdateSignaling aspect includes an after advice associated with a pointcut call(void Point.setX(int)) && within(Line), then the outline view indicates that the setX method in the Point class is *conditionally* extended by the after advice (Figure 9). Since the pointcut includes within(Line), the outline view shows that the behavior of setX is conditionally "extended by advice only if the caller is Line". The developers can see that the behavior of setX remains original if it is called from other classes than Line. If the combined pointcut is cflow, the outline view will show something like "extended if the thread is in the control flow of ..."

This visualization is different from AJDT. In AJDT, the influence of the within pointcut is equal between call and execution pointcuts. The within pointcut simply restricts the places indicated by arrow icons. In the case of the above pointcut, AJDT displays arrow icons only at the setX method calls that appear in the declaration of the Line class. AJDT does not show any information in the source code of the callee-side method setX.
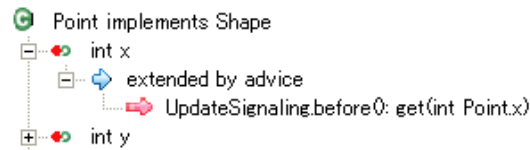
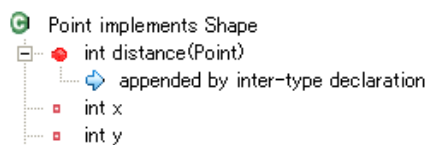Figure 10: There is a **before** advice associated with the **get** pointcut.

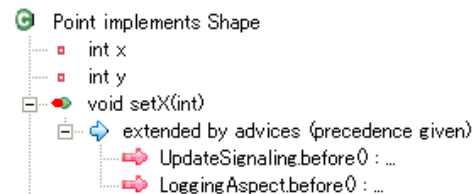Figure 11: An intertype declaration of the **distance** method

Figure 12: Two advices extend the **setX** method.

Other features:

The presentation of the **get** and **set** pointcuts in the outline view is similar to the **call** pointcut. In AspectJ, the join points selected by **get** and **set** pointcuts are field-access expressions at the accessor side (*i.e.* the caller side). Hence, AJDT shows an arrow icon at the line where the field is accessed. However, AspectScope interprets that an advice associated with a **get** or **set** pointcut extends the behavior of the target field. Figure 10 is an outline view presented by AspectScope. It illustrates the influence of an **UpdateSignaling** aspect that contains a **before** advice associated with a pointcut **get(int Point.x)**. Note that an arrow icon is shown below the **x** field in the **Point** class (*i.e.* at the target side) because the advice extends the behavior of the **x** field.

An aspect may include an intertype declaration. The methods and the fields appended by intertype declarations are also shown in the outline view. For example, Figure 11 indicates that an intertype declaration appends the **distance** method to the **Point** class.

If more than one advice extends a method or a field in an existing class, the outline view lists all the advices. If precedence rules are given by **declare precedence**, the multiple advice bodies extending the same method or field are listed in the execution order satisfying the given precedence rules (Figure 12). On the other hand, AJDT does not show the execution order of multiple advices.

Limitation:

AspectScope does not support all the language constructs of AspectJ. For example, AspectScope does not show any information of advice if the pointcut associated with that advice is the **handler** pointcut. The **handler** pointcut selects join points
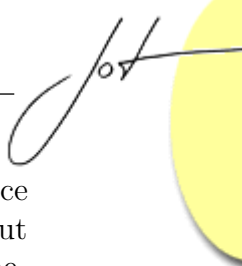
that represent the time when an exception is caught by a catch clause. An advice associated with this pointcut cannot be regarded as an extension to a method but it should be regarded as an extension to a try-catch statement. It is a more fine-grained extension and thus the visualization by AJDT would be more appropriate than AspectScope. Otherwise, it might be regarded as an extension to the behavior of an exception class, or a subclass of Throwable, because the advice modifies how instances of a particular exception class is handled. This is an open question.

Refactoring revisited

In Section 2, we presented an example of the figure editor. See Figure 1. When refactoring, the developer who uses AJDT could not see that a call to the moveBy method in the Line class is also advised. To know this fact, she has to see the source code of UpdateSignaling aspect or use the Call Hierarchy view of Eclipse IDE to visit all the caller sites to moveBy, which is a manual whole-program analysis.

AspectScope provides better help than AJDT in this scenario of refactoring. When an experienced developer does this refactoring, what does she do first? Before she starts editing the program, she will first check the specifications of the moveBy method in Line, which she is going to modify for refactoring. She will look at the outline view shown by AspectScope to confirm whether or not the specifications of it is extended. Then she will also check the specifications of the setX and setY methods in Point because she will use them when modifying the body of moveBy. Again, she will look at the outline view shown by AspectScope. Note that AspectScope also shows the javadoc-style description of the specifications of a selected method such as setX (Figure 5). It also help the developer understand a crosscutting structure in the program. We will mention details of the javadoc-style description in the next subsection.

Since the views shown by AspectScope tell her that the methods are extended by an aspect, she will soon understand that the naive implementation causes redundant display updates (Figure 2). She will also understand that a call to the moveBy method is advised and thus calling moveBy causes five display updates in total. Therefore, before editing the source code of the moveBy method, she can know that she must also modify the change pointcut in the UpdateSignaling aspect. Note that AJDT does not show her that the setX and setY methods are advised until she actually edits the source code of the moveBy method. After she writes a method-call expression to setX in the body of moveBy, AJDT marks the expression with an arrow icon that indicates the setX method is advised.

AspectScope displays the influence of an aspect in the outline view of the *callee-side* classes even if the aspect selects *caller-side* join points by the call pointcut and so on. This is a simple idea but it helps developer's modular reasoning. In typical OOP, the callee-side outline view corresponds to a module interface. The visualization by AspectScope is to project AOP structure onto module interfaces of OOP, which developers are familiar with. This is why the influence of an aspect is
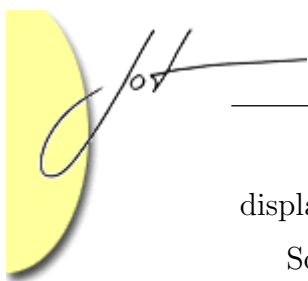
displayed in the outline view of callee-side classes.

Some readers might think that looking at the outline views of the setX and setY methods in our refactoring scenario is a sort of manual whole-program analysis. This is not true because the outline views are part of module interfaces. If developers are looking at only the implementation of a local module and the interfaces of other modules, then it can be said that they are doing local reasoning.

Javadoc pane

AspectScope provides not only the outline view but also the javadoc pane. The right pane of the AspectScope is the javadoc pane. It displays the javadoc comments of a selected member, such as a method and a field, in the outline view. The displayed javadoc comments are extracted not only from the source code of the selected member but also from aspects extending the member. Developers can read the comments to see details of the extension by the aspect, in other words, how the aspect affects the module interface.

The contents:

Figure 13 is a screen snapshot of the javadoc pane. It is displaying the javadoc comments of the setX method in the Point class. We assume that the pointcut and advice listed in Figure 14 were woven with the Point class. The displayed javadoc comments consist of four parts.

First, the text in (1) is constructed from the javadoc comments in the source code of the setX method. They describe the original behavior of the method. If any aspect is not deployed to extend the behavior of the setX method, the javadoc pane of AspectScope displays only this text.

The text in (2) to (4) is constructed from the source code of the aspect. If there are multiple aspects woven, the text is constructed for each aspect. The text in (2) describes that the behavior of the setX method is extended by an aspect. If the extension is conditional, the text in (2) also describes that condition. It is an English translation of the pointcut associated with the advice that extends the setX method. Note that it is not a naive translation of the pointcut expression, which is move() && within(Line). AspectScope expands a named pointcut such as move and removes unnecessary pointcuts. For example, call(void Shape+.moveBy(..)) is unnecessary because we are now interested only in the setX method; this pointcut never matches. call(void Shape+.set*(int)) is also redundant for the same reason. Since the method is setX in Point, AspectScope first expands wild-cards and displays an English translation of call(void Point.setX(int)). According to AspectJ's specification, the call pointcut selects join points by using the apparent type of a target object. Thus, AspectScope displays that the behavior of setX is extended only if the apparent type of the target is Point.
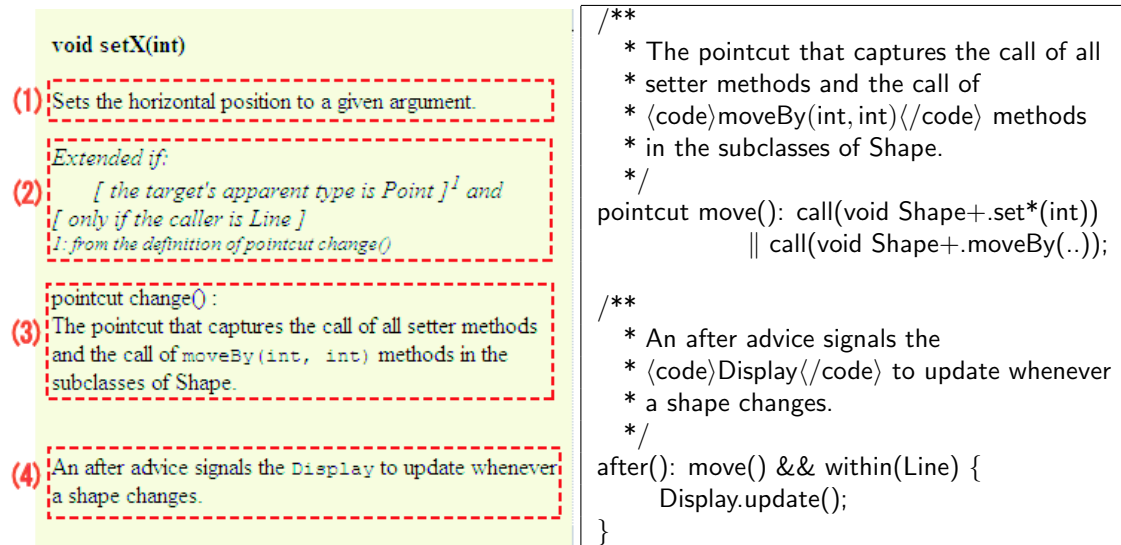
Figure 13: The javadoc pane for the setX method (the red dotted lines and text were drawn by the authors)

Figure 14: The definition of an advice and a pointcut

Recall that the **execution** pointcut uses the actual type of a target object. Thus, if the **call** pointcuts in Figure 14 were replaced with the **execution** pointcuts, then the text in (2) would not include the text related to the **execution** pointcut. AspectScope would never display "if the actual type is **Point**" because this phrase is redundant for describing when an advice extends the behavior of the **setX** method. When the **setX** method in **Point** is executed, the actual type of the target object must be **Point**! If there is no other pointcut remaining after unnecessary pointcuts are removed, AspectScope simply displays "Extended always" instead of "Extended if..." For example, if the **after** advice in Figure 14 were the following:
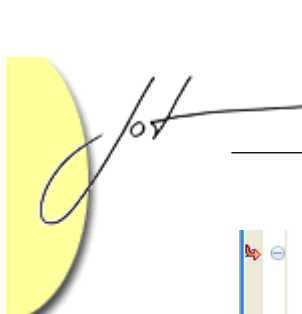
```
after():  execution(void Shape+.set*(int)) {
  Display.update();
}
```

then the text in (2) would be only "Extended always" because the behavior of the **setX** method is unconditionally extended by the **after** advice.

The text in (3) is constructed from the javadoc comments of the named pointcuts related to the **setX** method, such as the **move** pointcut. It is shown here for giving additional information on the condition of the extension by an aspect. Finally, the text in (4) is extracted from the source code of an advice that extends the behavior of the **setX** method. It describes details of that extension.

Figure 15: AJDT only shows that moveBy is advised.

Figure 16: The arrow icon indicating a join point shadow of cflowbelow

Wild cards:

When AspectScope shows a translation of pointcut in the javadoc view, it expands wild cards. For example, the wild cards in call(void Shape+.set*(..)) are expanded when AspectScope shows the javadoc comments for the setX method in Point. The result is call(void Point.setX(int)).

However, not expanding wild cards in a pointcut might be convenient, for example, when an aspect is homogeneous and implements a non-functional concern [1] such as access authentication. Developers might want to see the original pointcut containing wild cards. AspectScope always expands wild cards because it was designed for showing the module interface of each class under the deployment of aspects. It shows the javadoc comments for explaining how the behavior of a selected method or field is extended. If an aspect is homogeneous and a single advice body extends the behavior of multiple classes, this fact should be written in javadoc comments of that advice or its pointcut. Developers will be able to see the existence of the homogeneous aspect when they read the javadoc comments of one of the target method of that aspect through AspectScope.

4 EXAMPLES

In Section 3, we have already shown an example of AspectJ programming with AspectScope. We below show a few other examples.

Using the execution pointcut

AspectScope still provides a different visualization from AJDT's one even if the change pointcut in UpdateSignaling shown in Figure 1 is replaced with the following:

```
pointcut change():
   execution(void Point.setX(int))
   ‖ execution(void Point.setY(int))
   ‖ execution(void Shape+.moveBy(int,int));
```

---

[1]A non-functional concern is a concern independent of the application logic. Thus it is often commonly used among different applications.

Here, the **execution** pointcut is substituted for the **call** pointcut.

Since AspectScope deals with **call** and **execution** alike except translations, it shows the same outline view and the same javadoc comments as in the previous Section 3. Developers can still do modular reasoning.

On the other hand, AJDT only tells developers that the **moveBy** method is advised (Figure 15). They cannot see that the **setX** and **setY** are also advised. They must browse the source code of these methods. Some readers might think that browsing the source code of the methods is natural if the developers want to use them in the **moveBy** method. However, browsing the source code is not equal to looking at the module interfaces of the methods. It is rather looking at the internal implementation of a module and hence it is breaking the principle of information hiding. Of course, if appropriate javadoc comments are not provided by the developer, the users of AspectScope might also have to browse the source code of **setX** and **setY**. There is no serious difference between AJDT and AspectScope in that case.

Denotation of cflowbelow pointcut

To fix the problem of redundant display updates in Section 2, the **after** advice in the **UpdateSignaling** aspect must be updated to be this:

```
after() returning:  change() && !cflowbelow(change()) {
   Display.update();
}
```

AspectScope presents better representation after this update than AJDT.

As illustrated in Figure 16, AJDT displays an arrow indicating a join point shadow at the line where the **moveBy** method is called. However, this arrow icon does not show any extra information. Developers must click this icon to jump to the source code of the advice woven there. If they do not click, they cannot see the join points are selected by **cflowbelow**. On the other hand, AspectScope shows this fact within the javadoc comments of the **moveBy** method (Figure 17). The javadoc pane mentions that the **moveBy** method is extended "only if the apparent type is Line and not below the control flow of the call to **Line.moveBy(int, int)**". This fact is also shown in the outline view. Developers will be able to naturally see the exact effects of the **UpdateSignaling** aspect.

void moveBy(int, int)

moves this line by dx along the x axis and dy along the y axis

Extended if:
[the target's apparent type is Line]^1 and [not below
the control flow of the call to Line.moveBy(int, int)]
1: from the definition of pointcut change()

pointcut change() :
The pointcut that captures the call of all setter methods and the
call of moveBy(int, int) methods in the subclasses of Shape.

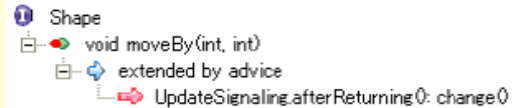An after advice signals the Display to update whenever a shape
changes.

Figure 17: The javadoc pane mentions cflowbelow.

Figure 18: The notification of moveBy method

Defining a new class implementing Shape

The final example is to define a new class implementing Shape for the figure editor. Let the name of the new class be Circle. The Shape interface is defined in Figure 1.

A developer who will define the Circle class would want to know that the moveBy method in Circle is extended by the UpdateSignaling aspect. However, AJDT does not tell her this fact until she defines the Circle class and writes client code. This is an example of undesirable obliviousness. She has to start writing the Circle class without knowing the extension by the aspect.

If the developer is an experienced engineer, she would first think that she should read the specifications of Shape. This is natural because she is going to define a class implementing the Shape interface. AspectScope helps such an experienced engineer. If she looks at the outline view that AspectScope shows for the Shape interface, she will notice that the moveBy method will be extended by the UpdateSignaling aspect (Figure 18). She can first know the extension by the aspect and then start writing the Circle class.

5   PRELIMINARY EVALUATION

To evaluate the usefulness of AspectScope, we used it for browsing the source program of ActiveAspect [5], which is a programming tool for AspectJ written by the third party. The program is written in AspectJ and it consists of 88 classes (10,683 lines) and 19 aspects (2,477 lines).[2]

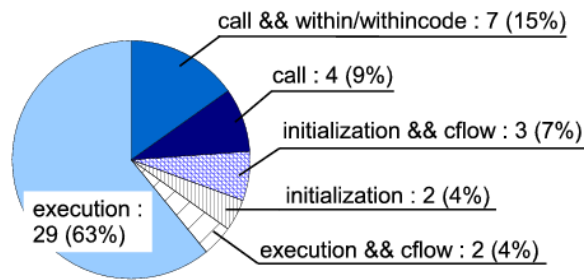[2]Since the original program has a few bugs, we did this study after fixing the bugs.

Figure 19: The analysis of the source program of ActiveAspect

The frequency of simple execution pointcuts

For the execution pointcut, the outline view of AspectScope is almost equal to the visualization by AJDT unless the execution pointcut is used with other pointcuts like cflow. Thus, if most pointcuts used in typical AspectJ programs are simple execution, the benefit of using AspectScope is relatively small.

Figure 19 shows the number of each pointcut designator used in the program of ActiveAspect. 67% of all the pointcuts are simple execution or initialization, which are expressed in the same way between AspectScope and AJDT. The join points selected by initialization are (part of) the execution of a constructor. The rest of the pointcuts are visualized by AspectScope in a different way from AJDT. They are pointcuts including call, within, withincode, or cflow.

The callee-side extension

When AspectScope visualizes call pointcuts, it interprets them as callee-side extensions although they select join points at the caller side (*i.e.* the client side). This is because it displays the effects of the pointcuts as changes of module interfaces, which are the outline view of the callee-side. We reviewed the program of ActiveAspect to examine whether or not this interpretation by AspectScope is acceptable for each pointcut.

A call pointcut combined with no other pointcut such as within selects method calls from any client site. Thus, an advice executed at these join points can be regarded as either a callee-side extension or a caller-side extension. There is no serious difference between the two interpretations; it is a natural interpretation that the advice is a callee-side extension.

A call pointcut combined with a target pointcut also selects method calls from any client site. The target pointcut restricts the actual type of the target object. An advice associated with such a call pointcut can be also regarded as a callee-side extension. An example we found in the program of ActiveAspect was the following:

Pointcut	Comments by the authors
execution(* SelectionOperator.apply()) && target(selector) && !cflow(execution(* MemberExpander.*(..)))	apply sets a flag of its target object except during the execution of a method in MemberExpander.
execution(* MemberEditPart.mousePressed(MouseEvent)) && target(ePart) && args(me) && !cflow(adviceexecution())	mousePressed performs an extra action depending on the state of the aspect instance. !cflow(..) is for avoiding infinite recursion.
call(ModelRelationship.new(..)) && !within(ModelRelationship)	The constructor displays an error message if it is not called from a singleton factory method.
(call(* ModelElement.addSourceRelationship(..)) || call(* ModelElement.addTargetRelationship(..))) && !within(ModelRelationship)	The methods add.. display a warning message if they are called from classes except ModelRelationship.
(call(* ModelElement.getModelCopy(..)) || call(ModelElement.new(..))) && !within(ProgramModel+) && !within(ModelElement+)	getModelCopy and the constructor display a warning message if they are called from classes except the specific classes.
call(* ModelRelationship.setHidden(boolean)) && args(boolean) && target(ModelRelationship) && !within(StickyRels)	!within(StickyRels) avoids the infinite recursive execution of this advice in StickyRels.
call(* IDrawableEntity.setLocation(..)) && target(ModelElement) && withincode(* ClassifierEditPart.createFigure(..))	setLocation performs an extra action if it is called from the createFigure method in ClassifierEditPart.
initialization(AggregateRelationship+.new(..)) && target(ModelRelationship) && cflow(execution(void AbstractMembersRule.apply()))	The constructor sets a flag of the created object if it is called during the execution of the apply method.
initialization(AggregateRelationship+.new(..)) && target(ModelRelationship) && cflow(execution(void AbstractRelationsRule.apply()))	the same as above except apply is a method in not AbstractMembersRule but AbstractRelationsRule. This pointcut is used by two advices, which set a different flag.

Table 1: All pointcuts declared in the source code of ActiveAspect

```
after(Object modelObj, AbstractEditPart editPart):
   call(void EditPart+.setModel(Object)) && args(modelObj)
                                   && target(editPart) {
   ((IDrawableEntity)modelObj).setEditPart(editPart);
}
```

This advice makes a reverse link from the argument to the target object when the setModel method is called on an AbstractEditPart object. AbstractEditPart is a subclass of EditPart. It is natural interpretation that this advice extends the callee-side setModel method.

Interesting pointcuts with respect to interpretation are call pointcuts combined with within, withincode, or cflow. The execution and initialization pointcuts with cflow are also interesting. Because within and cflow restrict caller-side contexts, we thought that it might be less natural to interpret the advices combined with such pointcuts as callee-side extensions. However, as we listed in Table 1, we could not find any advices that *must* be interpreted as caller-side extensions. For example, the following code is one of the advices that it is least natural to interpret as callee-side extensions:

```
after(ModelElement elt):
    call(* IDrawableEntity.setLocation(..))  && target(elt)
        && withincode(* ClassifierEditPart.createFigure(..))  {
    setLocation(elt);
}
```

This advice changes the behavior of the **setLocation** method only when it is called from the **createFigure** method. **setLocation(elt)** calls a method declared in the aspect including the above advice. It performs the dedicated action for playing the demo of the software. Since the behavior depends on the caller site, it is somewhat inappropriate to interpret this advice as a pure callee-side extension. However, this interpretation is still acceptable.

In summary, the interpretation by AspectScope was not a serious problem in our preliminary case study. However, the program of ActiveAspect does not include non-functional homogeneous aspects, such as logging and authentication, which may not fit the interpretation by AspectScope. Although Apel et al. also reported that such aspects are not frequently used [2][3], we need further study on this topic.

6   RELATED WORK

AspectScope dynamically generates module interfaces according to current deployment of aspects. The generated interfaces are not statically determined ones. This idea was borrowed from aspect-aware interfaces [10]. Thus, we can say that AspectScope is a programming tool that realizes the idea of aspect-aware interfaces in AspectJ. However, interpreting aspects as extensions to callee-side classes is a unique feature of AspectScope. In the original article of aspect-aware interfaces, the interpretations of the **call**, **get**, and **set** pointcuts are open questions (in Section 4.2 of [10]). They even suggest interpreting an aspect including those pointcuts as a *caller*-side extension.

Our callee-side interpretation of AOP is similar to the Classbox approach [3] although Classboxes are not AOP language constructs. Classboxes are modules that can provide a custom interface to selected clients. Although Classboxes provide better information hiding and modularity, AOP languages provide better expressiveness for describing conditional extensions (or custom interfaces in the terminology of Classboxes).

Active models [5] is another approach to represent a crosscutting structure better than AJDT. ActiveAspect, which is their tool based on the active models, presents a node-and-link diagram representing an interesting slice of the crosscutting structure of an AspectJ aspect. Although ActiveAspect and our AspectScope share the same

---

[3] In [2], most aspects are implemented by mixins. They correspond to AspectJ's advices associated with the **execution** pointcut.
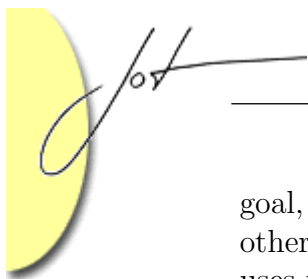
goal, ActiveAspect's approach is to visualize join points selected by aspects. On the other hand, our AspectScope visualizes module interfaces extended by aspects. It uses traditional tree-based representation.

Another approach to address the drawbacks of the obliviousness property is to introduce language constructs into AOP languages. There have been several constructs proposed on this approach: for example, open modules [1, 16] and XPIs (crosscut programming interfaces) [8]. Their idea is to let developers declare a module interface for pointcuts. They must explicitly specify selectable join points from external clients so that the fragile pointcut problem [11] can be avoided. The developers can take care of those selectable join points when they modify the implementation of the module. A disadvantage of this approach is that developers must anticipate join points that will be selected by aspects deployed in future. Anticipating all necessary join points in advance is difficult. Otherwise, developers must manually update module interface whenever new join points must be selectable. The approach of AspectScope is to visualize currently selected join points and hence it complements the approach of open modules and XPI.

7   CONCLUDING REMARKS

AspectScope performs a whole-program analysis of AspectJ programs and visualizes the result so that developers can understand their program behavior with local reasoning. It displays the module interfaces extended by aspects under current deployment.

A unique idea of AspectScope is to interpret an aspect as an extension to the callee-side (target-side) class even if the aspect includes the call pointcut. This enables expressing the effects of aspects through module interfaces. Developers thereby do AOP by using their OOP experiences of modular programming, in particular, modular extensions to classes by virtual classes [13], mixin-layers [18], nested inheritance [14, 15], and so on.

On the other hand, AspectScope is inappropriate for aspects that this interpretation does not fit although such aspects would be not many. For such aspects, we should switch tools to AJDT. A tracing aspect for debugging and a transaction aspect often fall into this category. Such aspects interpret join points as *events* that triggers the execution of advice code [6, 4, 1]. For example, they use a call pointcut for executing an advice in the middle of the method body of the caller. Such an advice is independent of the *callee* side and it is used only for extending the behavior of the *caller-side* (client-side) method. It should be regarded as a caller-side extension. Although those aspects are also significant applications of AOP, the influence of the aspects on module interfaces is difficult to express.

ACKNOWLEDGMENTS

REFERENCES

[1] Aldrich, J., "Open Modules: Modular Reasoning About Advice," in *ECOOP 2005*, LNCS 3586, pp. 144–168, Springer-Verlag, 2005.

[2] Apel, S. and D. Batory, "When to Use Features and Aspects?: A Case Study," in *Proc. of the 5th Int'l Conf. on Generative Programming and Component Engineering (GPCE '06)*, pp. 59–68, ACM Press, 2006.

[3] Bergel, A., S. Ducasse, and O. Nierstrasz, "Classbox/J: Controlling the Scope of Change in Java," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 177–189, 2005.

[4] Cilia, M., M. Haupt, M. Mezini, and A. Buchmann, "The Convergence of AOP and Active Databases: Towards Reactive Middleware," in *Proc. of Generative Programming and Component Engineering (GPCE '03)*, pp. 169–188, Springer-Verlag, 2003.

[5] Coelho, W. and G. C. Murphy, "Presenting Crosscutting Structure with Active Models," in *Proc. of 5th Int'l Conf. on Aspect-Oriented Software Development (AOSD 2006)*, pp. 158–168, ACM Press, 2006.

[6] Douence, R. and M. Südholt, "A model and a tool for Event-based Aspect-Oriented Programming (EAOP)," Technical report 02/11/INFO, École des Mines de Nantes, 2002.

[7] Filman, R. E. and D. P. Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness," in *Aspect-Oriented Software Development* (R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, eds.), pp. 21–35, Addison-Wesley, 2005.

[8] Griswold, W. G., M. Shonle, K. Sullivan, Y. Song, N. Tewari, Y. Cai, and H. Rajan, "Modular Software Design With Crosscutting Interfaces," *IEEE Software*, vol. 23, pp. 51–60, January/February 2006.

[9] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," in *ECOOP 2001 – Object-Oriented Programming*, LNCS 2072, pp. 327–353, Springer, 2001.
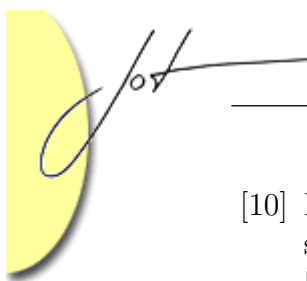
[10] Kiczales, G. and M. Mezini, "Aspect-Oriented Programming and Modular Reasoning," in *Proc. of the Int'l Conf. on Software Engineering (ICSE'05)*, pp. 49–58, ACM Press, 2005.

[11] Koppen, C. and M. Stoerzer, "PCDiff: Attacking the Fragile Pointcut Problem," in *Proc. of European Interactive Workshop on Aspects in Software (EIWAS'04)*, 2004.

[12] Masuhara, H., G. Kiczales, and C. Dutchyn, "Compilation Semantics of Aspect-Oriented Programs," in *Proc. of Foundations of Aspect-Oriented Languages Workshop*, AOSD 2002, pp. 17–26, 2002.

[13] Mezini, M. and K. Ostermann, "Integrating Independent Components with On-Demand Remodularization," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 52–67, 2002.

[14] Nystrom, N., S. Chong, and A. C. Myers, "Scalable Extensibility via Nested Inheritance," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 99–115, 2004.

[15] Nystrom, N., X. Qi, and A. C. Myers, "J&: Software Composition with Nested Intersection," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 21–36, 2006.

[16] Ongkingco, N., P. Avgustinov, J. Tibble, L. Hendren, O. de Moor, and G. Sittampalam, "Adding open modules to AspectJ," in *Int'l Conf. on Aspect Oriented Software Development (AOSD'06)*, pp. 39–50, ACM Press, 2006.

[17] Rashid, A., "Aspects and Evolution: The Case for Versioned Types and Meta-Aspect Protocols." Keynote talk at 3rd ECOOP Workshop on Reflection, AOP, and Meta-Data for Software Evolution, 2006.

[18] Smaragdakis, Y. and D. Batory, "Mixin Layers: an Object-Oriented Implementation Technique for Refinements and Collaboration-based Designs," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 215–255, 2002.

[19] Steimann, F., "The Paradoxical Success of Aspect-Oriented Programming," *ACM SIGPLAN Notices*, vol. 41, no. 10, pp. 481–497, 2006.

[20] The Eclipse Foundation, "AspectJ Development Tools (AJDT)." http://www.eclipse.org/ajdt.

ABOUT THE AUTHORS
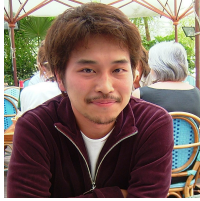
Michihiro Horie is a master student at Tokyo Institute of Technology, Japan. He received his B.S. in computer science from Tokyo Institute of Technology in 2006. His interests are programming languages, especially aspect-oriented programming languages.

Shigeru Chiba is an associate professor at Tokyo Institute of Technology, Japan. He received his PhD degree in computer science from The University of Tokyo in 1996. He is interested in programming languages, middleware, and operating systems. His current research topic is aspect-oriented software development.