

Semantics-Based Crosscutting in AspectJ™

Gregor Kiczales,[†] Jim Hugunin,[‡] Mik Kersten,[‡] John Lamping,[‡]
Cristina Lopes[‡] and William G. Griswold^{††}

Introduction

A central issue in aspect-oriented programming is the design of mechanisms for specifying the crosscutting. Early systems used explicit enumeration of names of messages or methods[...]. Recent versions of AspectJ go beyond this by allowing a crosscut to be described based on semantic properties of the operations involved. We intend for future versions of AspectJ to go even farther in this direction. In this short paper we describe the full range of crosscut specification mechanisms we currently plan. The paper is intended to serve as a foundation for discussion of this important issue.

Implementation and detailed rationale are outside the scope of this short paper.

Crosscut Definition Mechanisms in AspectJ

Crosscuts in AspectJ are defined in terms of key events in the execution of Java programs.² These events include invocation of a message³ from the caller object, reception of a message at the called object, execution of a method, and signaling and handling of exceptions. So, for example, a program can define a crosscut that includes the execution of any `int getSize()` method in a particular package. AspectJ makes it possible to define actions that should be executed at certain well-defined times in relation to such events. These times are before the event, after the event, and around the event. The after case is further divided into after if the event terminates normally, after if the event throws certain exceptions, or after in all cases. Around the event means that the crosscut action will have explicit control over whether the event should be allowed to proceed and in what dynamic context it does proceed.

[†] University of British Columbia (gregor@cs.ubc.ca). [‡] Xerox PARC. [†] Purple Yogi. ^{††} University of California, San Diego (currently visiting Xerox PARC). This work was partially supported by the Defense Advanced Research Projects Agency under contract number F30602-C-0246.

² The AspectJ semantics is not defined in terms of a weaver copying code into the beginning and end of methods. A particular implementation of AspectJ might do this a variety of ways—it might copy code (as in a pre-processor), it might insert dispatches (as in a fast compiler), it might use other techniques. The work of the aspect weaver component of the language implementation is “to coordinate the crosscutting”; it was never intended to be so particular as to mean “to copy code around”.

³ In this paper the term “message” is used to refer to the combination of a method name, its result type and the types of its parameters. In JVM terminology this is a `NameAndType`.

Simple Name-Based Crosscuts

The simplest way to specify a crosscut is to explicitly name the messages in the crosscut.

```
01 class logPoint1 extends Logger {
02     crosscut gets(): Point & (int getX() | int getY());
03     crosscut sets(): Point & (void setX(int) | void setY(int));
04     crosscut accesses(): getterCut() & setterCut();
05     crosscut constructions(): Point & new(..);
06
07     static advice(): accesses() {
08         after {
09             log.write("A point was accessed.");
10         }
11     }
12
13     static advice(): constructions() {
14         after {
15             log.write("A point was constructed.");
16         }
17     }
18 }
```

This class consists of four crosscut declarations and two advice declarations. Line 2 declares a crosscut named `gets()` with a signature of `()`. The part to the right of the colon designates the events that comprise the crosscut. This designator reads as “all events in Point objects” and (“all messages `int getX()`” or “all messages `int getY()`”), which amounts to whenever a Point object receives one of the two access messages. In line 3 the `sets()` crosscut is comprised of reception of the two setter messages. Line 4 shows that crosscut designators can also refer to other crosscuts; the `accesses()` crosscut is comprised of all the events in the previous two crosscuts. Line 5 shows the syntax for designating constructor invocations; the `constructions()` crosscut is comprised of invocations of the no arguments constructor for Points.

Lines 7-11 define after advice on the `accesses` crosscut. The effect is that after point objects receive `int getX()`, `int getY()`, `void setX(int)` or `void setY(int)` message, they will write “a point was accessed” to a log.

Crosscut designators can include not just multiple events but also events in multiple classes—it is called crosscutting after all—so for example the `gets()` crosscut defined above could instead be:

```
02b crosscut gets():
03b     (Point & (int getX() | int getY() | new())) |
04b     (Line4 & (Point getP1() | Point getP2() | new()));
```

Of course in this case the message in the advice “A point operation happened” would need to be changed.

⁴ Assume that a Line aggregates two Points in the natural way.

Crosscut Signatures

In AspectJ, a crosscut can expose certain values involved in the events. Exposing them means advice defined on the crosscut have access to those values. A crosscut can, for example, expose the object receiving the message, one or more of the parameters of the message, or the return value.

```
20 class logPoint2 extends Logger {
21   crosscut gets(Point p) :      p & (int getX() | int getY());
22   crosscut sets(Point p, int v): p & (void setX(v) | void setY(v));
23
24   static advice(Point p): gets(p) {
25     after {
26       log.write("Getter on " + p + ".");
27     }
28   }
29   static advice(Point p, int v): sets(p, v) {
30     before {
31       log.write("Setter on " + p + " to " + v + ".");
32     }
33   }
```

The way this works is that the formal parameters of the crosscut (i.e. “(Point p)” in line 21) indicate the values the crosscut will expose. Then, in the designator, the formal parameter names are used in place of type names to denote which values are exposed. So “p” in the right hand side of line 21 means two things: (i) “all events in Point objects” because the type of p is Point in the crosscut formal, and (ii) expose the Point object itself.

The crosscut can also expose the value returned by an event, but because it can only do so to after and around events we use a different syntax for this.

```
40 class logPoint3 extends Logger {
41   crosscut gets (Point p) returns int: p & (int getX() | int getY());
42
43   static advice (Point p) returns int: gets(p) {
44     after (int result) {
45       log.write("Getter on " + p + " returned " + result + ".");
46     }
47   }
48 }
```

Wildcards

Crosscuts designators can have wildcards in them. A wildcard can appear anywhere a type name can appear, in this case the wildcard syntax is ‘*’. A wildcard can also appear anywhere a list of type names can appear, this is marked with ‘..’. The following are examples of crosscut designators with wildcards:

```
02b crosscut gets():      * & (int getX() | int getY());
02c crosscut gets(): Point & ( * getX() | * getY());
02d crosscut gets():      * & ( * getX() | * getY());
```

```
02e crosscut gets(): Point & * get*();
```

Line 02b uses a wildcard in a type name position. This designates invocations of all messages `int getX()` or `int getY()` on all types in the current package.

In line 02c, `*` is used in the return type position of the message signatures. This means match messages named `getX` and `getY`, and which accept no arguments, regardless of their return type.⁵

Line 02d shows a more reasonable use of `*` in the result type position. Because this now can match more than one type, it can match more than one message. So, assuming the package has `Point` and `FloatPoint` types, with the natural accessors, this will designate invocations of four different messages.

Line 02e uses a currently unimplemented feature, the ability to use wildcards within the name of a message. The meaning of this line would be to match invocations of any message on `Point` that accepts no arguments, returns any result type and whose name starts with “get”.

```
50 crosscut allMessages(): * & * *(..);  
51 crosscut allConstructors(): * & new(..);
```

Line 50 defines a crosscut comprised of all message invocations on all types in the current package. Line 51 is for all constructor invocations on all types in the current package.

A package can be explicitly specified by preceding a type name with it. Packages can be wildcarded using `..` as follows:

```
52 crosscut allMessages(): mypackage..* & * *(..);
```

The syntax ‘`mypackage..*`’ means any type in `mypackage` or any of its sub-packages.

Sub-Method Granularity

The calls crosscut modifier designates the caller side of message and constructor invocations. So, for example, the crosscut

Simple Property-Based Crosscuts

In this section we show some of the simplest semantics based crosscuts supported by AspectJ. These crosscuts are all defined in terms of statically observable properties of messages. For example, line 53 defines a crosscut comprised of invocations of all public messages in the `mypackage` package hierarchy.

⁵ Since Java doesn’t allow overloading on result type, this can only match two messages (one `getX()` and one `getY()`). So in this case the use of `*` doesn’t mean “match multiple” it means “I don’t know what the return type is”. (It could mean “I’m too lazy to put the return type in”, but that would be sloppy programming.)

```

53 crosscut publicCalls ():
54     mypackage..* & (public * *(..) | public new(..))

```

Advice defined on this cut could take actions required whenever execution enters or exits the package. So, for example, the following code would log every error thrown by a public message to its caller.

```

60 class LogPublicErrors {
61     crosscut publicCalls():
62         mypackage..* & ( public * *(..) | public new(..) );
63
64     static advice(Object o): publicInterfaceCut(o) {
65         catch (Error e) {
66             log.write(e);
67         }
68     }

```

This simple example shows some of the power of semantics based crosscuts. Because the program crosscuts the public interface without enumerating it, then if the public interface changes this code will not have to be edited. Without semantics based crosscuts the programmer would have to remember to update the list of messages in the public interface. (Without AOP at all it is worse of course, the programmer has to remember to add code to each method.)

Other examples of common semantics-based crosscuts include:

```

mypackage..* & public new(..)           invocations of public constructor in mypackage
mypackage..* & public !static * *(..)  invocations of public non-static messages in mypackage

```

We are also beginning to consider crosscuts specified in terms of dynamic properties of the program's execution. The first of these makes it possible to denote as being comprised of events within the dynamic extent (aka control flow) of an invocation. This can be used to designate invocations that are dynamically in service of certain other particular operations.

One way this can be used is to discriminate between internal and external calls to an interface. In the following code the getters for Line and Polygon themselves call the getters for Point. The gets() advice crosscuts all getter invocations. But the second one rules out invocations that are themselves within the control flow of another getter invocation.

```

70 class Counter {
71
72     static double getterInvocations = 0;
73     static double topLevelGetterInvocations = 0;
74
75     crosscut gets():
76         ( Point & ( int getX() | int getY() )) |
77         ( Line & ( Point getP1() | Point getP2() | float getLength() )) |
78         ( Polygon & ( Point[] getPoints() |

```

```

79         Line[] getLines() |
80         float getCircumference()));
81
82     crosscut topLevelGets(): getters() & !cflow(getters());
83
84     static advice(): getters() {
85         before {
86             getterInvocations++;
87         }
88     }
89
90     static advice(): topLevelGetters() {
91         before {
92             topLevelGetterInvocations++;
93         }
94     }
95 }

```

Summary

AspectJ includes a variety of mechanisms for specifying crosscuts. In addition to the now familiar name-based mechanisms, AspectJ makes it possible to define crosscuts based on static and dynamic semantic properties of the program.

The ability to specify crosscuts in semantic terms is intended to provide several kinds of benefits. First we believe that programs will be easier to read. The idea here is that “mypackage.* & public * *(..)” immediately comes across as the public message invocations of mypackage, in a way that a long list of message names would not. Second we believe programs will be less error prone in initial development. The idea here is that it is easy to leave a method out of the long list! Third we believe that programs will be more robust during evolution because a crosscut specified in semantic terms automatically tracks changes to the rest of the program. If you add a public method it will automatically be comprised in the crosscut, you won’t have to remember to add it to the long list.