Journal of Universal Computer Science, vol. 14, no. 21 (2008), 3498-3516
submitted: 16/4/08, accepted: 5/6/08, appeared: 1/12/08 © *J.UCS*

Controlling Aspect Reentrancy[1]

Éric Tanter
(PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile, Chile
etanter@dcc.uchile.cl)

Abstract: Aspect languages provide different mechanisms to control when an aspect should apply based on properties of the execution context. They however fail to explicitly identify and cleanly capture a property as basic as that of reentrancy. As a result, aspect developers have to resort to low-level and complex pointcut descriptions that are error prone and hamper the understandability of aspect definitions. We analyze the issue of aspect reentrancy, illustrate how current languages fail to properly support it, and define a new linguistic construct to control aspect reentrancy. Considering aspect reentrancy from the start in the design of an aspect language simplifies the task of aspect programmers by raising the level of abstraction of aspect definitions.
Key Words: Aspect-oriented programming, pointcut language, aspect reentrancy
Category: D.1.m, D.3.3

1 Introduction

In the pointcut-advice (PA) family of aspect-oriented languages [Wand et al. 2004], as represented by AspectJ [Kiczales et al. 2001], crosscutting behavior is defined in aspects, *i.e.* modules that encompass a number of pointcuts and advices. Points during execution at which advices may be executed are called (dynamic) join points. A pointcut identifies a set of join points, and an advice is the action to be taken at a join point matched by a pointcut. A particular area of research is that of dynamic join point selection, for which many mechanisms exist [Tanter 2007, Avgustinov et al. 2007], for instance, based on lexical scope and dynamic context like `within` and `cflow` in AspectJ.

Aspect developers are frequently exposed to the issue of reentrant aspect application [AspectJ List 2008], that is, when an aspect matches join points that are triggered by its own execution or a recursive base program. As a result, advices are executed multiple times, or the program enters infinite loops. Although most of the time reentrant application of aspects is a programming error, developers are left with low-level idioms to address this issue [Bodden et al. 2006]. These idioms, which rely on combining lexical and dynamic pointcut designators, are error prone and hamper the understandability of aspect definitions. This indicates a limitation of current aspect language design.

[1] This work is partially funded by FONDECYT Project 11060493.

We identify and illustrate different cases of aspect reentrancy and their associated idioms; this leads us to identify a flaw in the semantics of mainstream AspectJ implementations (Section 2). Our contribution is to make reentrancy an explicit concept in the design of an aspect language (Section 3). We introduce a new default semantics for pointcut-advice aspect languages, which avoids unwanted reentrant aspect application, and a dedicated construct to control reentrancy at a fine-grained level when needed, at the appropriate level of abstraction. We illustrate our proposal and formulate its semantics. Section 4 discusses related work and Section 5 concludes.

1.1 Running Example

We illustrate our argument using a simple example familiar to the aspect community: that of `Point` objects that can be moved. Class `Point` is defined in a very standard manner:

Program 1 Definition of points.

```
public class Point {
  int x,y;
  public int getX() { return x; }
  public void setX(int x) { this.x = x; }
   //similarly for getY and setY
  public void moveTo(int x, int y){ setX(x); setY(y); }
  public String toString(){ return getX() + "@" + getY(); }
}
```

We introduce an aspect whose role is to emit a notification whenever a `Point` object is "active", that is, one of its method is executing on it. A first naive definition of this aspect is as follows:

Program 2 Activity notifier aspect.

```
public aspect Activity {
  before() : execution(* Point.*(..)){
    System.out.println("point active");
} }
```

The following section discusses the problems of this first definition, coming from reentrant application of the aspect. We also consider progressive refinements of the aspect functionality. At each step, we discuss how to address the reentrancy issue, whenever possible. The solutions make use of pointcut designators that are used to control the scope of an aspect based on lexical and dynamic

contexts. To this end, aspect languages support different kinds of scoping constructs [Tanter 2007, Tanter 2008]. Although the argument is formulated in the context of AspectJ, it applies to any aspect language in the pointcut-advice family [Wand et al. 2004].

2 Reentrant application of aspects

Whenever a join point is matched by a pointcut, the associated advice is executed. During the process of pointcut matching, advice activation, or subsequent base program execution, a join point may be generated that is matched by the same aspect. This potentially reentrant application of aspects can manifest either as an advice executing several times instead of once, or worse, as an infinite loop.

To better understand the phenomena at stake, we distinguish three kinds of reentrancy, depending on the code that may emit the join point that potentially triggers the reentrancy; we call such a join point a *reentrant join point*. This categorization of the kinds of reentrancy is useful to study the different remedies that can be devised, as well as for the formulation of our proposal.

2.1 Base-triggered reentrancy

Base-triggered reentrancy refers to cases where the base program is responsible for the reentrancy. This can be due to base-level recursive calls, for instance, if the aspect advises a recursive method. Another example occurs when a base method whose execution is advised by an aspect invokes another method that is also advised by the same aspect. This case of reentrancy is well-known in the aspect community as it was used since the early days of AOP as a motivating example for control-flow related pointcut designators.

Consider the activity notifier aspect of Program 2. When `moveTo` is called on a `Point`, the aspect applies and issues a notification. However, `moveTo` in turns calls the coordinate setter methods of the point, resulting in execution join points that are again matched by the aspect. As a result, we get three notifications instead of one. Similar cases can occur with nested invocations of constructors or super calls.

Antidote. To avoid base-triggered reentrancy, programmers usually resort to an idiom that consists in extending the definition of the aspect to ensure it only matches on top-level execution join points. This discrimination relies on execution flow information. AspectJ supports two pointcut designators for this: `cflow` discriminates join points that are *in* the control flow of a given pointcut, while `cflowbelow` matches join points *below* that control flow.

In the case we are considering, the antidote consists in filtering out reentrant join points by discriminating them based on the fact that they are *below* the control flow of an already-matched join point (Program 3). Note that it is compulsory to introduce the named pointcut `active` in order to refer to it in the `cflowbelow` expression (otherwise the full definition of `active` has to be manually inlined). This is because AspectJ supports neither recursive pointcut definitions nor a "this pointcut" variable.

Program 3 Avoiding base-triggered reentrancy.

```
public aspect Activity {
  pointcut active(): execution(* Point.*(..));
  before() : active() && !cflowbelow(active()) {
    System.out.println("point active");
} }
```

2.2 Advice-triggered reentrancy

Advice-triggered reentrancy occurs when the execution of an advice itself produces reentrant join points. This kind of reentrancy manifests as infinite advice execution. This phenomena is well-known in the reflection community as the infinite meta-regression problem [Kiczales et al. 1991, Chiba et al. 1996, Denker et al. 2008]. It has also been identified in the case of AspectJ by Bodden and colleagues [Bodden et al. 2006], as discussed in Section 4.

There are two variants of advice-triggered reentrancy. The execution of the advice can produce reentrant join points that are either *(a)* join points of the base program, *e.g.* by invoking a method on an object; *(b)* join points of its own execution, *e.g.* by invoking one of its methods, or by matching the advice execution itself.

Let us consider an extension of the `Activity` aspect such that it prints out the point object that is active (Program 4). Here reentrancy occurs because the advice prints the currently-executing `Point` object, which implies a call to its `toString` method. The execution of the `toString` method is in turn matched by the pointcut. As a result, the program enters an infinite loop.

Antidote. It is important to understand here that the `!cflowbelow(active())` condition does not help in avoiding reentrant execution of the aspect. This is because the reentrant join point, the execution of `toString`, does *not* occur in the control flow of the execution join point that triggered the advice: indeed, the advice is executing *before* the method executes

Program 4 Printing the active point.

```
public aspect Activity {
  pointcut active(): execution(* Point.*(..));
  before(Point p) : active() && this(p) && !cflowbelow(active()) {
    System.out.println("point active: " + p);
} }
```

on the `Point` object. Rather, the reentrant join point is in the control flow of the advice execution itself.

There are different idioms that one can apply to avoid this kind of reentrancy. The AspectJ programming guide recommends using the `within` pointcut to exclude join points that are in the lexical scope of the aspect[2]. However, this is a very narrow antidote that does not apply in our case, because the reentrant join point does not occur *lexically* in the aspect, but rather occurs in the control flow of the advice execution. One could therefore use the `!cflow(adviceexecution())` idiom. While this solution works, it is too extensive because `adviceexecution` does not discriminate between the different aspects that may exist in a system. A more precise solution is to use `!cflow(within(A))`, where `A` is the aspect that reentrantly applies [Bodden et al. 2006]. This well-known idiom excludes all join points that occur in the control flow of any join point that is lexically in the aspect `A`. Program 5 shows our fixed example.

Program 5 Avoiding advice-triggered reentrancy.

```
public aspect Activity {
  pointcut active(): execution(* Point.*(..));
  before(Point p) : active() && this(p)
    && !cflowbelow(active())
    && !cflow(within(Activity)) {
    System.out.println("point active: " + p);
} }
```

2.3 Pointcut-triggered reentrancy

Most aspect languages today support dynamic pointcut designators. These pointcuts generally cannot be fully evaluated at compile time. Depending on the semantics of the aspect language, runtime evaluation of dynamic pointcuts may generate reentrant join points.

[2] http://www.eclipse.org/aspectj/doc/released/progguide/pitfalls-infiniteLoops.html

AspectJ supports several dynamic pointcut designators, such as runtime type checks of join point attributes, and `if` pointcuts. Runtime type checks cannot cause reentrancy because `instanceof` checks are not join points in the AspectJ join point model. However, the `if` pointcut can contain any valid Java boolean expression. This expression can invoke any accessible method, and can also access join point attributes provided that they are exposed by the pointcuts (for instance to check that the argument of a join point matches a given criteria).

As an example, consider an extension of the `Activity` aspect such that only points that are located within a certain area are monitored (Program 6). We assume that point objects have an `isInside` method that takes an area object as parameter. The invocation of `isInside` in the `if` pointcut leads to an execution join point of the denoted method, which causes the pointcut evaluation to reenter. The pointcut evaluation process therefore enters in an infinite loop, resulting in a stack overflow error.

Program 6 Pointcut-triggered reentrancy.

```
public aspect Activity {
  Area area =  /* obtain reference to user-selected area */
  pointcut active(): execution(* Point.*(..));
  before(Point p) : active() && this(p)
    && !cflowbelow(active())
    && !cflow(within(Activity))
    && if(p.isInside(area)) {
    System.out.println("activity in " + p);
} }
```

Antidote. One would expect the previous antidote –which rules out join points that are in the control flow of a join point occurring lexically in the aspect– to suffice here. Indeed, the call to `isInside` in the `if` pointcut of Program 6 does occur lexically in the aspect, and triggers the execution join point that provokes non-termination. So, the condition `!cflow(within(Activity))` should make it possible to avoid pointcut-triggered reentrancy as well.

However, it turns out that the semantics of AspectJ as implemented by the two main AspectJ compilers, `ajc` and `abc` [Avgustinov et al. 2006], do not match with what one would expect. These compilers both adopt the same strategy with respect to evaluation of `if` pointcuts: join points that are *lexically* in the `if` pointcut definition are *hidden*. Join points that are in the execution flow of these join points are, however, visible. This means that in Program 6 the *call* join point to `isInside` is not visible, and hence cannot be used to rule out the subsequent *execution* join point of `isInside`.

Program 7 Avoiding pointcut-triggered reentrancy.

```
public aspect Activity {
  Area area =  /* obtain reference to user-selected area */
  pointcut active(): execution(* Point.*(..));
  before(Point p) : active() && this(p)
    && !cflowbelow(active())
    && !cflow(within(Activity))
    && if( checkArea(p) ) {
    System.out.println("activity in " + p);
   }
  boolean checkArea(Point p) { return p.isInside(area); }
}
```

A solution to this problem can be obtained by moving the actual body of the `if` pointcut to a separate method in the aspect, as illustrated on Program 7. This refactoring "works" because while the call to the newly-introduced `checkArea` method remains hidden, its execution is fully visible. This means that the execution of `isInside` now occurs in the control flow of a join point lexically present in the aspect body. Alternatively, we could have moved the call to `isInside` to the advice body –although that would defeat the purpose of having `if` pointcuts–.

We claim that the semantics currently implemented by main AspectJ compilers is flawed for the following reasons:

- The semantics of a program should not be affected by moving the body of an `if` pointcut to a separate method (as illustrated in Program 7), or to the advice itself. Hiding the join points that are lexically in `if` pointcuts violates this equivalence.
- Hiding join points can result in inconsistencies. For instance, if another aspect needs to intercept all calls to `isInside` (*e.g.* to intervene on the area that is used), it will not be aware that the method is called when the `if` condition of `Activity` evaluates.

To sum up, as a consequence of the multiple ad hoc and idiomatic solutions we had to use in order to avoid all three kinds of reentrancy, our aspect definition has become fairly complex and obscure (Program 7), in particular if compared with an ideal definition that would match our original intensions:

```
public aspect Activity {
  Area area = ...;
  before(Point p) : execution(* Point.*(..)) && this(p)
                    && if(p.isInside(area)) {
    System.out.println("activity in " + p);
} }
```

2.4 Over-strict control of reentrancy

After having illustrated the different kinds of reentrancy and the way they can be addressed, we would like to raise the issue of over-strict control of reentrancy. We now turn our attention to the fact that for our activity aspect, execution of methods on different `Point` objects should be different activities.

Consider the aspect definition of Program 3: it uses a `!cflowbelow` condition in order to avoid base-triggered reentrancy. Let us now consider an extension of the `Point` class with an `attract` method, that moves the argument point next to the receiver point:

```
void attract(Point p){
  p.moveTo(this.x+1, this.y);
}
```

If we run the following program, with the `Activity` aspect, we would expect to be notified of the activity of both `p1` and `p2`:

```
Point p1 = new Point(1,2);
Point p2 = new Point(3,4);
p2.attract(p1);
```

However, we only get notification of the activity of `p2`. Why is it so? When `attract` executes, it calls `moveTo` on `p1`; however, the execution of `moveTo` occurs below the control flow of the `attract` execution join point, therefore the `cflowbelow` condition discards it. If we remove the `cflowbelow` condition, we are back to the beginning, facing unwanted base-triggered reentrancy.

Therefore we are unable to respect the semantics of the `Activity` aspect: to notify of each activity starting in point objects. The root of the problem here is that `cflowbelow` is a property that holds for a whole thread of execution, regardless of the object that is currently executing.

If we were to specify a solution to this problem, we would have to introduce a whole infrastructure to keep per-point thread-local state. For the interested reader, the code of a possible implementation is given on Program 8. It is clear that this solution is not at the right level of abstraction. It basically boils down to reimplementing an object-local version of `cflowbelow` by hand.

3 Supporting Aspect Reentrancy

The previous section clearly illustrates that the issue of reentrant aspect application, although conceptually simple to apprehend, is not well served by current aspect language design. Out of the three cases of reentrancy we discussed, two require different idioms built out of low-level mechanisms, and one is simply not solvable with current semantics, unless the program is refactored.

Program 8 Definition of the `Activity` aspect with per-object thread-local state.

```
public aspect Activity {
  ThreadLocal<Integer> Point.count = new ThreadLocal<Integer>(); // 0
  void Point.inc() {  /* increment count */;}
  void Point.dec() {  /* decrement count */;}
  int Point.count() { return (int) count.get();}
  boolean Point.active() { return count() > 0; }

  pointcut active(): execution(* Point.*(..));
  pointcut active-nr(Point p):
    active() && this(p) && !cflow(within(Activity));

  before (Point p) : active-nr(p) {
    if(p.count() == 0) System.out.println("activity in " + p);
    p.inc();
  }
  after (Point p) : active-nr(p) {
    p.dec();
} }
```

Furthermore, the idioms we have presented also have their limitations. First, having to combine them makes pointcut definitions complex. Second, they require references to static elements like named pointcuts and aspect type names, creating unnecessary dependencies that hamper reuse and extensibility.

The claim of this paper is that this issue deserves a unified language mechanism to make avoiding reentrant application of aspects a triviality, while allowing particular reentrancy scenario expressible. This section exposes our proposal.

3.1 Language design choice

From a language design point of view, we have to make a choice between proposing a backward-compatible extension to AspectJ that allows for aspect reentrancy control, and formulating a new semantics for AspectJ that breaks backward compatibility but promotes reentrancy control at the core of the pointcut language.

The first alternative consists in adding a pointcut called `reentering` that is true whenever any kind of reentrancy is involved. This provides a simple way for programmers to say "do not match if the join point is reentering", and thereby avoid all cases of reentrancy at once. With such a design, programmers have to explicitly state when reentrancy should be avoided, as in Program 9. (Note that this program does not solve the per-object reentrancy issue.)

Although preserving backward compatibility, this approach requires explicit thinking about reentrancy. We agree with Bodden *et al.* that reentrant application of aspects is almost always a programming error [Bodden et al. 2006]. It is very easy to find numerous cases of unwanted reentrancy, while actual exam-

Program 9 Using the `reentering` pointcut.

```
public aspect Activity {
  Area area = ...;
  before(Point p) : execution(* Point.*(..)) && this(p)
    && if(p.isInside(area)) && !reentering() {
    System.out.println("activity in " + p);
} }
```

ples where matching reentrant join points is needed are fairly scarce. Section 2.4 presented one of these.

Contrasting with the current AspectJ semantics, we opt to promote aspect stability above all, by making avoiding matching reentrant join points a *default in the language*. Instead, we provide a means to recover the possibility to match reentrant join points when needed, by means of a dedicated pointcut designator, exposed below in Section 3.2. It will become clear to the reader that adopting one or the other approach does not impact on the essence of our proposal, whose semantics is described in Section 3.3.

3.2 A pointcut for dealing with reentrancy

Our proposal makes non-matching of reentrant join points a default. This means that the correct definition of the `Activity` aspect is a straightforward expression of our design intensions, as shown on Program 10. Reentrant join points caused by execution of setters while moving a point, the verification that the point is located in the given area in the `if` pointcut, and the invocation of `toString` as part of the advice, are simply not matched by the aspect.

Program 10 Non-reentrant aspects as a language default.

```
public aspect Activity {
  Area area = ...;
  before(Point p) : execution(* Point.*(..)) && this(p)
    && if(p.isInside(area)) {
    System.out.println("activity in " + p);
} }
```

There are indeed cases where an aspect needs to match *some* reentrant join points, based on a certain scope. For instance, in our example (Sect. 2.4) we want the `Activity` aspect to support object-level reentrancy: the aspect must notify of the activity of each single object, even if it is activated below the control flow of the activity of another one.

pointcut definition	reentrant join points matched?
`pc()`	never
`reentrant(pc())`	always (AspectJ)
`reentrant[this](pc())`	if currently-executing objects differ
`reentrant[target](pc())`	if target objects differ
`reentrant[class-of(this)](pc())`	if classes of currently-executing objects differ

Table 1: Controlling reentrancy: some strategies.

To this end, we introduce a higher-order pointcut designator called `reentrant`, that can be used to define precise reentrancy strategies, as shown on Table 1. For instance, Program 11 refines the previous program by supporting object-level reentrancy: the original pointcut is wrapped with `reentrant[this]`. Here, `this` is a *function* that specifies the scope of reentrancy: it extracts the currently-executing object at a join point, which is then used to discriminate reentrancy.

Program 11 Object-level reentrancy.

```
public aspect Activity {
  Area area = ...;
  before(Point p) : reentrant[this](
   execution(* Point.*(..)) && this(p) && if(p.isInside(area))) {
    System.out.println("activity in " + p);
} }
```

Since the scope is defined by a function, it is possible to use any computable criteria to define the scope of reentrancy. For instance, one can abstract over individual objects when defining the scope of reentrancy, *e.g.* class-level reentrancy can be defined using `reentrant[class-of(this)]`. Other scoping semantics are possible, for instance to control reentrancy by object groups, like in the POM language for coordination of parallel activities [Caromel et al. 2008], or to discriminate reentrant join points based on their target object, arguments, lexical scope, etc.

Finally, if the scope function is omitted, then all reentrant join points are matched: this is the closest to the current AspectJ semantics[3].

[3] The difference being that AspectJ hides join points that are lexically in `if` pointcuts (which we consider a flaw, as discussed at the end of Section 2.3).

3.3 Semantics

We now give a definition of the semantics of our proposal. It boils down to defining what it precisely means for a join point to be considered reentrant, for a given aspect, and a given scope. We first introduce a simple model of execution traces with join points and aspects, and illustrate the different cases of reentrancy. We then present our extension to this model to support both global and scoped reentrancy.

3.3.1 Preliminaries: join points and aspects

An execution trace is modeled as an ordinal tree whose nodes are join points. A join point consists of a kind, some data (like the name of the called/executing method, the currently-executing object, the target object, the arguments), and the previous join point. Attributes of a join point are accessed with a standard dot notation, like $jp.kind$, $jp.this$, and $jp.parent$. A join point is an abstraction of the control stack [Wand et al. 2004]; $stack(jp)$ denotes the set of prior join points of jp.

An aspect is modeled as a set of pointcut-advice pairs. Pointcuts and advices are modeled as functions, following [Dutchyn et al. 2006]. This means that equality of pointcuts and equality of advices are defined as equality of functions: two functions are deemed equal if they have the same body and close over the same lexical environment.

Illustration. Figure 1 shows the execution trace of evaluating `p.moveTo(a,b)`. A node is a join point and a vertex represents the parent join point relation. For call join points, we include the name of the called method and the target object, while for execution join points, we include the name of the executing method and the currently-executing object (this). A grey join point denotes a join point matched by the `Activity` aspect.

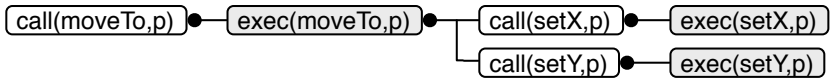

Figure 1: Execution trace with join points.

We can visualize on Figure 1 the occurrence of base-triggered reentrancy: the setter execution join points are matched by the aspect, and are in the control flow of a matched join point (the execution of `moveTo`).

Figure 2 extends the representation with the execution of the advice of `Activity`, triggered by the matching of `exec(moveTo,p)`. An advice execution

join point is generated (as in AspectJ), following which a call to `toString` occurs. This call results in an execution join point that is also matched by the aspect, resulting in an infinite loop. This is a case of advice-triggered reentrancy.

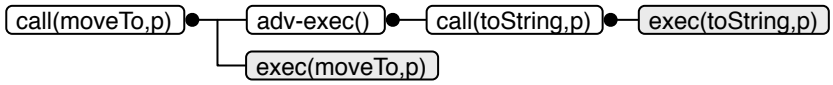

Figure 2: Advice execution and advice-triggered reentrancy.

Figure 3 illustrates the case of pointcut-triggered reentrancy, as in Program 6. The call to `isInside` that occurs within the `if` triggers a reentrant execution join point `exec(isInside,p)`. We can see the crux of the problem: the reentrant join point does not have any relation in its stack history that can be used to discriminate it in order to avoid reentrancy. Indeed, its parent join point, `call(isInside,p)` is *hidden* so it appears as if it is a join point completely unrelated to the aspect matching process. It is also not in the control flow of another execution join point. So the infinite loop cannot be avoided.

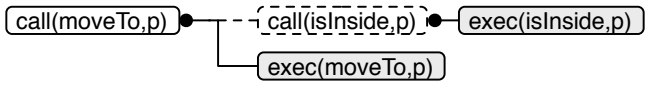

Figure 3: Pointcut matching and pointcut-triggered reentrancy.

3.3.2 Global reentrancy

Let us first look at the semantics of a reentrant join point, when we only consider global reentrancy. To detect base-triggered reentrancy, the semantics of `cflowbelow` is therefore sufficient.

To detect advice-triggered reentrancy we need a join point for advice execution. This is semantically more appropriate than the lexical-scope discrimination used in the AspectJ idiom `!cflow(within(A))`. Indeed, in AspectJ and most languages, aspect instances are objects upon which methods can be called, independently of any advice execution. The typical AspectJ idiom is therefore flawed in this respect. AspectJ already includes an advice execution join point, but as we already explained it is too general as it represents *any* advice execution. To avoid this pitfall in our model, advice execution join points are parametrized by the advice that is (about to be) executing.

Definition 1 Global reentrancy

A join point jp is reentrant for the pointcut-advice pair (pc, adv)
$\iff \exists\, jp_2 \in stack(jp)$, such that one of the following properties holds:

(BR) $match(pc, jp_2)$

(PR) $jp_2.kind = PC\_MATCH \ \wedge \ jp_2.pc = pc$

(AR) $jp_2.kind = ADV\_EXEC \ \wedge \ jp_2.adv = adv$

Finally, to detect pointcut-triggered reentrancy there is no option but to introduce a *pointcut matching* join point. This allows us to discriminate join points that are in the control flow of a pointcut evaluation. A pointcut matching join point is also parametrized by the pointcut that is being evaluated.

Definition 1 formalizes the semantics of a globally reentrant join point. It basically says that for a join point to be reentrant for a given pointcut-advice pair, it has to be either base reentrant (BR) –this is equivalent to `cflowbelow`–, pointcut reentrant (PR) –meaning it is in the control flow of the evaluation of the pointcut–, or advice reentrant (AR) –meaning it is in the control flow of the execution of the advice. Note that this definition ensures that two advices (*e.g.* before and after) can be associated to the same pointcut and both execute without being ruled out for reentrancy concerns (because the advice comparison of (AR) will fail).

Figure 4 illustrates how this applies to our example. Thanks to the presence of the **pc-match** and **adv-exec** join points, the execution join points of `isInside` and `toString` are straightforward to discriminate (using (PR) and (AR) respectively).

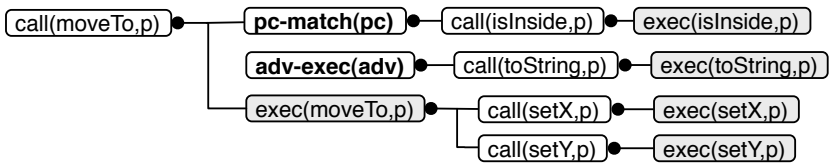

Figure 4: Introducing **pc-match** and **adv-exec** to determine reentrancy.

3.3.3 Scoped reentrancy

Let us now look at the semantics of a reentrant join point for a finer-grained, scoped notion of reentrancy, such as per-object reentrancy.

As briefly mentioned in Section 3.2, the scope of reentrancy is defined in our model by a context function of type $CTX = JP \rightarrow Value$. That is, a context function takes a join point and returns one of its (computed) characteristics. For instance:

- The function to extract the currently-executing object: $this = jp \rightarrow jp.this$
- The function to extract the target object: $target = jp \rightarrow jp.target$
- We can also define abstraction mechanisms, eg: $class\_of : CTX \rightarrow CTX$, so we can use $class\_of(this)$ as a context function and get class-based reentrancy.

We use context functions to relate two different join points at the time where the question is raised of whether or not the latter is reentrant. If the context function that defines the scope of reentrancy yields the same value for both join points, then the latter is reentrant. Therefore, we have to be able to relate a potentially reentrant join point with the join point that originally triggered the aspect application process, called *root* join point (`exec(moveTo,p)` in our example).

We hence extend the model so that the **pc-match** and **adv-exec** join points reference the root join point. This way we can retrieve this join point later when examining the stack history of a join point that may cause reentrancy. The updated illustration is given on Figure 5. The root join point (denoted **rjp** on the figure) is kept within the **pc-match** and **adv-exec** join points.

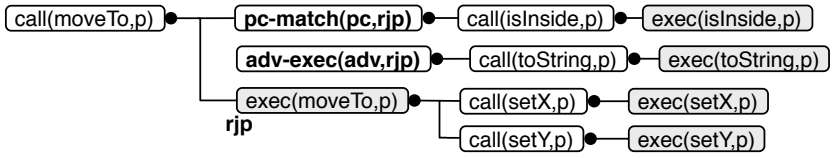

Figure 5: Including current join point **jp** in **pc-match** and **adv-exec** to determine scoped reentrancy.

With these elements at hand, Definition 2 refines the previous definition of join point reentrancy to formalize scoped reentrancy[4]. The only difference is the

[4] Definition 1 is equivalent to Definition 2, where ctx is a constant-valued function, *i.e.* such that $\forall x \forall y, ctx(x) = ctx(y)$.

Definition 2 Scoped reentrancy

A join point jp is reentrant for the pointcut-advice pair (pc, adv) for scope ctx $\iff \exists\, jp_2 \in stack(jp)$ such that one of the following properties holds:

(BR) $match(pc, jp_2) \ \wedge \ ctx(jp) = ctx(jp_2)$

(PR) $jp_2.kind = PC\_MATCH \ \wedge \ jp_2.pc = pc \ \wedge \ ctx(jp) = ctx(jp_2.jp)$

(AR) $jp_2.kind = ADV\_EXEC \ \wedge \ jp_2.adv = adv \ \wedge \ ctx(jp) = ctx(jp_2.jp)$

conjunction of a condition on the context function. Note that in the case of (PR) and (AR), we use the fact that the **pc-match** (resp. **adv-exec**) join point jp_2 holds the original join point, and apply the context extractor function on that join point. In our example (Figure 5), this makes it possible to relate the p of the $exec(isInside, p)$ join point to the p of the not-yet matched join point jp that is the root cause of this pointcut evaluation.

3.4 Implementation

With the definition of the semantics of our proposal for reentrancy control in an aspect language, the question of how to implement it can be raised. Looking at the definitions of reentrant join points, we can see that the only reasoning needed is very similar to that of a control flow pointcut, possibly with context exposure. In addition, there exists a wide body of knowledge on how to efficiently implement control flow based matching (*e.g.* [Avgustinov et al. 2005]). This is much simpler than the general case of trace monitoring, where relations between join points cross over stack boundaries [Avgustinov et al. 2007]. Our work is concerned first and foremost with the semantic issues in aspect languages; we deliberately leave the door open to the study of how to efficiently support the proposed semantics in AspectJ.

4 Related Work

The issues of infinite regression have been widely explored in the reflection community [Kiczales et al. 1991, Chiba et al. 1996], and continue to be so [Denker et al. 2008]. The focus is on handling the regression caused by meta-level entities. This corresponds to advice-triggered reentrancy in our terminology. Although base-triggered reentrancy does apply, it is generally not considered. Reflective architectures do not have pointcuts, so pointcut-based reentrancy does not apply.

POM [Caromel et al. 2008] is a domain-specific aspect language for coordination of parallel activities. POM gives control over reentrancy of requests in

schedulers. This corresponds to addressing base-triggered reentrancy explicitly in advices. POM is not a full-fledged aspect language, so pointcut-based reentrancy does not manifest.

A semantics for pointcuts and advices is given in [Wand et al. 2004]. No particular provision is made to deal with reentrancy beyond the presence of an advice execution join point and the control flow primitives, which make it possible to express the AspectJ idioms we have discussed. The model does not consider `if` pointcuts, so the necessity of a pointcut matching join point to address pointcut-based reentrancy is not identified.

To our knowledge Bodden *et al.* [Bodden et al. 2006] are the first to explore a solution to the problem of recursive advice application beyond AspectJ idioms. Their work is very related to work on metalevel architectures, because they recognize the meta-ness of advices and provide a means to specify the exact level at which join points are potentially matched. Although very satisfying from a theoretical viewpoint, this approach puts high demands on programmers. Our proposal is less expressive (it does not distinguish exact levels of reentrancy), but it promotes simplicity. Finally, stratified aspects only address advice-triggered reentrancy, while we uniformly deal with the three kinds of reentrancy.

In recent work, we propose expressive scoping semantics for dynamically-deployed aspects, called deployment strategies [Tanter 2008]. While reentrancy is related to scoping, deployment strategies are independent from the actual pointcut language semantics. In contrast, this work addresses reentrancy at the level of the pointcut language, and even requires the introduction of new kinds of join points.

5 Conclusion

Current aspect languages fail to explicitly identify and cleanly capture aspect reentrancy, leaving developers with low-level, error-prone, and invasive means to address this issue. We propose to treat aspect reentrancy as a first-class concern in the design of an aspect language, by avoiding reentrant aspects by default, and providing an appropriate language construct to specify precise and structured reentrancy semantics when required. Our proposal is consistent with respect to join point visibility, and requires the introduction of pointcut matching and advice execution join points. These join points are primarily meant to be used by the aspect runtime to properly deal with reentrancy. The language design we propose promotes a safe default for aspects, making them more stable and simple. We believe that this work is a valuable step towards pointcut definitions that are more faithful to the intensions of the programmer.

Acknowledgment

We are grateful to Eric Bodden and Guillaume Pothier for their help in clarifying our initial intuition. Eric rightly insisted for the adoption of non-reentrancy as a default semantics, at the cost of backward compatibility. We also thank the anonymous reviewers for their comments.

References

[AspectJ List 2008] AspectJ List (2008). The AspectJ users mailing lists archives. http://www.eclipse.org/aspectj/userlists.php.

[Avgustinov et al. 2005] Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. (2005). Optimising AspectJ. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI 2005)*, pages 117–128, Chicago, USA.

[Avgustinov et al. 2006] Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. (2006). abc: an extensible AspectJ compiler. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *Lecture Notes in Computer Science*, pages 293–334. Springer-Verlag.

[Avgustinov et al. 2007] Avgustinov, P., Tibble, J., and de Moor, O. (2007). Making trace monitors feasible. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2007)*, pages 589–608, Montreal, Canada. ACM Press. ACM SIGPLAN Notices, 42(10).

[Bodden et al. 2006] Bodden, E., Forster, F., and Steimann, F. (2006). Avoiding infinite recursion with stratified aspects. In *Proceedings of Net.ObjectDays 2006*, Lecture Notes in Informatics, pages 49–54. GI-Edition.

[Caromel et al. 2008] Caromel, D., Mateu, L., Pothier, G., and Tanter, É. (2008). Parallel object monitors. *Concurrency and Computation: Practice and Experience*, 20(12):1387–1417.

[Chiba et al. 1996] Chiba, S., Kiczales, G., and Lamping, J. (1996). Avoiding confusion in metacircularity: The meta-helix. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer-Verlag.

[Denker et al. 2008] Denker, M., Suen, M., and Ducasse, S. (2008). The meta in meta-object architectures. In *Proceedings of TOOLS Europe*, Lecture Notes in Business and Information Processing, Zurich, Switzerland. Springer-Verlag. To appear.

[Dutchyn et al. 2006] Dutchyn, C., Tucker, D. B., and Krishnamurthi, S. (2006). Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239.

[Kiczales et al. 1991] Kiczales, G., des Rivières, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. MIT Press.

[Kiczales et al. 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. (2001). An overview of AspectJ. In Knudsen, J. L., editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary. Springer-Verlag.

[Tanter 2007] Tanter, É. (2007). On dynamically-scoped crosscutting mechanisms. *ACM SIGPLAN Notices*, 42(2):27–33.

[Tanter 2008] Tanter, É. (2008). Expressive scoping of dynamically-deployed aspects. In *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*, pages 168–179, Brussels, Belgium. ACM Press.

[Wand et al. 2004] Wand, M., Kiczales, G., and Dutchyn, C. (2004). A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910.