Issues on Observing Aspect Effects from Expressive Pointcuts

Hidehiko Masuhara and Tomoyuki Aotani

Graduate School of Arts and Sciences, University of Tokyo
masuhara@acm.org, aotani@graco.c.u-tokyo.ac.jp

Abstract. This paper discusses issues on interactions of aspects with expressive pointcuts. Since expressive pointcuts specify join points based on the results of program analysis, they should be carefully designed in order to analyze effects of aspects and their precedence correctly. We show examples in which aspects with expressive pointcuts interact, and point out the required properties to those pointcuts for correct aspect interaction. We also briefly present our approach to satisfy those properties in our SCoPE compiler, which supports expressive pointcuts within AspectJ language.

1 Introduction

Aspect-oriented programming (AOP) languages allow more than one aspects affect at the same point. On one hand, this enables modularization of cross-cutting concerns that overlap the same points. On the other hand, this causes aspect interactions. For example, assume we apply two aspects to a method, namely a tracing aspect, which records method executions, and a memoization aspect, which reuses previously calculated results of methods. Then those two aspects applied to a program interact because the program could behave differently depending on the precedence of aspects. If the tracing aspect precedes the memoization, all invocations of the memoized methods are recorded. If the memoization precedes the tracing, invocations with memoized arguments are not recorded.

One of the important challenges in designing AOP languages is to support *expressive pointcuts*, which specify join points based on high-level information on a program and its execution, such as a method calling context, predicted behavior and information flow. Those pointcuts can express programmers' intention more directly and can improve robustness of aspects against program evolution.

Although there are quite a few studies on the languages that support new expressive pointcuts [1–7], interactions of aspects with expressive pointcuts are not seriously considered. Since expressive pointcuts depend on the behavior of a program, they should take effects from aspects into account. Moreover, the effects from aspects should be carefully chosen in order to isolate effects from irrelevant aspects.

This paper focuses on the expressive pointcuts that are based on static program analysis, such as predicted control flow [1] and coding style rules [4]. Although the following discussion assumes the pointcut and advice mechanism in AspectJ language [8], it can be applied to other AOP mechanisms like the inter-type declarations and to other AOP languages.

2 Expressive Pointcuts based on Static Analysis

Before discussing aspect interactions, we first review two examples that use expressive pointcuts based on static analysis.

2.1 Expressive Pointcuts

Expressive pointcuts specify join points based on high-level information on a program and its execution [1, 9–13] in order to address the fragile pointcut problem. Those pointcuts exploit more stable information than signatures so that trivial program modifications do not change the set of join points specified by the pointcuts. In addition, expressive pointcuts enable the programmers to describe their intention in a more declarative manner. Further discussion on the advantages of expressive pointcuts can be found in other literatures [7].

A typical aspects with expressive pointcuts are defined in conjunction with regular pointcuts such as `call` and `set`. The regular pointcuts with wildcards first specify kind and static scope of join points (e.g., "any field sets to figure objects"). Then the expressive pointcuts restrict to the join points with certain properties (e.g., "sets to such fields that are used for displaying figures").

Although this paper discusses by using an AOP language that supports user-defined expressive pointcuts [7], the raised issues are applicable to built-in expressive pointcuts. The main reason to take user-defined expressive pointcuts as examples is that they concisely illustrate implementations of the expressive pointcuts. Usefulness of user-defined expressive pointcuts can be found in other literatures [2–7].

2.2 Examples

Predicted Control-Flow. `pcflow` is a hypothetical expressive pointcut that predicts control flow [1]. It matches join points that are potentially reachable from join point shadows matching the given pointcut.

Fig. 1 is an example given by Kiczales [1] that demonstrates usage of `pcflow` by using a figure editor program with a display updating concern. The three classes on the left hand side represent figures, and the aspect on the right hand side updates a display whenever a program changes any visual property of a figure.

The `displayState` pointcut represents field gets under the predicted control flow of—i.e., that are potentially performed during—the `draw` methods of the `Fig`'s subclasses. It therefore represents field gets from `Point.x`, `Point.y`,

```
abstract class Fig {
  abstract void draw();
}
class Point extends Fig {
  int x,y;
  void draw() {
    Display.plotXY(x,y);
  }
}
class Line extends Fig {
  Point p1,p2;
  void draw() {
    Display.line(p1.x, p1.y,
                 p2.x, p2.y);
  }
}
```

```
aspect DisplayUpdating {
  //predict control flow of
  //Fig.draw() and find fields
  //referenced in that control flow
  pointcut* displayState():
    pcflow(
      execution(void Fig+.draw()))
    && get(* Fig+.*);

  //sets to such fields require
  //display update
  after(): set(<displayState()>) : {
    Display.update();
  }
}
```

Fig. 1. Figure classes (left) and display updating aspect with `pcflow` (right) [1].

`Line.p1` and `Line.p2`. The pointcut `set(<displayState()>)` in the advice declaration matches all sets to the fields represented by `displayState`. Consequently, when a program modifies a visual property (e.g.,, `x` of a `Point` object), the aspect calls `Display.update()` to redraw the modified figure.

The `DisplayUpdating` aspect with the `pcflow` pointcut is more robust against program evolution than the one defined by enumerating concrete field signatures. Assume we modify the figure editor to support colors by adding a `color` field to the `Fig` class, and performs "`Display.setColor(color);`" at the beginning of each `draw` method. The `DisplayUpdating` aspect updates the display when the program sets to the `color` field because the `displayState` pointcut automatically includes the `color` field. If the advice declaration uses concrete field names instead of the `pcflow` pointcut, the modification would require to change the `DisplayUpdating` aspect as well.

Side-Effect Freedom. `isSideEffectFree` is also a hypothetical expressive pointcut that analyzes whether a method accesses a global state. It matches a join point when the code reachable from the shadow of the join point does not perform field sets and gets.

Fig. 2 shows an example use of the `isSideEffectFree` pointcut. The `Memoization` aspect is to optimize method executions by reusing the result of a previous execution with the same argument. The `memoPoint` matches any method executions that takes an integer as an argument and returns an integer as a result, but restricted to, thanks to the `isSideEffectFree` pointcut, the methods that performs no side-effecting operations.

The use of the `isSideEffectFree` pointcut makes the aspect more safe because it automatically excludes methods that access a global state. Otherwise, the programmer who changes a method has to make sure that the method can or can not be memoized, and update the pointcut if needed.

```
aspect Memoization{
  Map<Signature,Map<int,int>> caches = ...;
  pointcut memoPoint(int key): execution(int *(int)) && args(key)
                              && isSideEffectFree();
  int around(int key): memoPoint(key) {
    // obtain the hash table associated to the current method
    cache = caches.get(thisJoinPoint.getSignature());
    if (!cache.contains(key))      //if the argument is not in the table,
      cache.put(key,proceed(key)); //run the method and record the result
    return cache.get(key);         //otherwise, return recorded result
  }
}
```

Fig. 2. An aspect to reuse the results of non-side-effecting methods.

```
aspect Coloring {
  Color Fig.color;                      //each figure has a color
  before(Fig fig): this(fig)
      && execution(void Fig+.draw()) {  //before drawing a figure,
    Display.setColor(fig.color);        //change the current color
  }                                     //to the figure's
}
```

Fig. 3. An aspect to add colors to figures.

2.3 Languages that Support User-Defined Expressive Pointcuts

The topic of the paper—semantics of expressive pointcuts—is not only for language designers, but also for advanced programmers. This is because studies on AOP system enable advanced programmers to define new pointcut primitives by themselves. Such systems include an extensible compiler [14], extensible languages [2, 3] and our proposed SCoPE compiler, which supports expressive pointcuts through the `if` pointcut in AspectJ [7].

3 Required Properties of Expressive Pointcuts for Aspect Interactions

Expressive pointcuts with static analysis make aspect interactions more complicated because the programs that are analyzed by the expressive pointcuts can be affected by aspects themselves. When expressive pointcuts are poorly designed and implemented, aspects that ought to interact could no longer interact.

Below, we summarize the required properties of expressive pointcuts for enabling correct aspect interactions.

3.1 Aspect Visibility

Assume we have a `Coloring` aspect (shown in Fig. 3) that allows individual figures in Fig. 1 to have different colors. The aspect implements this feature by declaring a `color` field in `Fig` class, and letting `draw` methods change the drawing color to the figure's.

Here, the `DisplayUpdating` aspect should aware of the effects from the `Coloring` aspect because the latter aspect lets the `draw` methods perform field gets from additional fields. This means that the `pcflow` pointcut should analyze the program including the effects of the `Coloring` aspect. Even it might sound trivial, it is not so actually because the pointcut needs a compiled (or woven) program to analyze, while the compiled program is generated only after matching the pointcut.

To generalize, in order to make aspects interact correctly, the following property should be satisfied:

> *Required Property 1: Effects of aspects should be visible from the analyses of expressive pointcuts.*

The property is also rationalized by the semantics of the other features in most AOP languages that let aspects observe effects of aspects. An advice declaration observes advice execution join points and join points during advice execution. Also, as pointed out by Havinga, et al. [15], the inter-type declaration mechanism can also observe effects of other inter-type declarations.

When an expressive pointcut of an advice declaration observes the effects of the advice itself, it is not trivial to satisfy the property as pointed out by Klose et al. [16]. We will discuss this in Sec. 4.1.

3.2 Delimited Target of Analysis

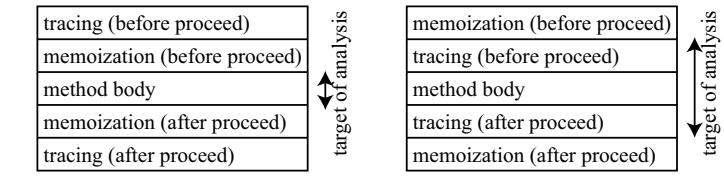
Assume a method that accesses no global state, and the `Memoization` aspect (Fig. 2) and a tracing aspect (whose definition is not given in this paper but obvious) applied to the method. The `Memoization` aspect stores the resulting values of method executions for later uses if they have no (observable) side-effect (as determined by using the `isSideEffectFree()` pointcut), and the tracing aspect records method executions into a log file.

Depending on the precedence of those two aspects, the `Memoization` aspect works differently by the following reasons:

1. When the `Memoization` aspect precedes the tracing aspect, memoization should not be performed because, from the viewpoint of the `Memoization`, the execution of the method is side-effecting as it includes the behavior of the tracing aspect.
2. When the tracing aspect precedes the `Memoization` aspect, memoization should be performed because the method execution from the viewpoint of the `Memoization` has no side-effect.

This means that the expressive pointcut `isSideEffectFree()` used in the `Memoization` should match the join points when the `Memoization` is less precedent to the tracing aspect. In other words, the `isSideEffectFree()` pointcut should be aware of the precedence of aspects.

Note that we do not claim which aspect should precede the other, nor that the expressive pointcut should control the precedence of aspects. Rather, our



(a) when tracing precedes memoization (b) when memoization precedes tracing

Fig. 4. Code layouts of the woven method.

claim is that, given aspect precedence (which can be controlled by using `declare precedence` in ApsectJ), the `isSideEffectFree()` pointcut should give different results.

In order to let the memoization aspect work only when the tracing aspect has precedence, the `isSideEffectFree` pointcut should observe the effects from aspects *including aspect precedence*.

One of the reasonable solutions to this problem is to let `isSideEffectFree()` analyze the code only reachable from `proceed` in the advice body. Depending on the aspect precedence, the code layout of the woven method will look like Fig. 4 (a) and (b). By delimiting the targets of analysis with the code fragments reachable from the `proceed` (as shown by the arrows on the side of the boxes in the figure), the pointcut can analyze the program with correctly including or excluding the effects of other aspects.

To generalize, the following property should be satisfied:

> *Required Property 2: Expressive pointcuts should be able to delimit the target of their analysis to the code fragments that are actually executed when aspect precedence is taken into account.*

4 Aspect Interaction with SCoPE Compiler

SCoPE is a compiler that supports expressive pointcuts within AspectJ language [7]. In SCoPE, an expressive pointcut is realized by writing a program analysis as a condition of an `if` pointcut, which is a build-in construct in AspectJ, with the help of reflection API and bytecode analysis tools. Although the `if` pointcut is dynamically tested with the standard AspectJ compilers, SCoPE evaluates those pointcuts when they do not access runtime information, leaving no runtime overheads.

Fig. 5 and Fig. 6 show implementations of the expressive pointcuts `pcflow` and `isSideEffectFree` shown before. In Fig. 5, `pcflowGet` traverses the instructions that are reachable from the given method, and returns true when there is an instruction that references the field that is set by the current join point. In Fig. 6, `isSideEffectFree` traverses the instructions that are reachable from the `proceed` expression in the advice body, and returns false when there is any instruction that accesses a global variable.

```
aspect DisplayUpdating {
  static boolean pcflowGet(JoinPoint jp, String entry){
    for(Method m: implsOf(entry))       //for each method implementation,
      for(Instruction i: m.instructions)//for each instruction,
        if (i is a field get &&         //return true if it gets from the same
            i.hasSameSignature(jp))     //field to the current join point.
          return true;
        else if (i invokes method m1 && //recursively check the invoked
                 pcflowGet(jp, m1))     //method
          return true;
    return false;                       //false when no such field get
  }
  after(): set(* Fig+.*) &&
      if(pcflowGet(thisJoinPoint,
         "execution(void Fig+.draw())")): {
    Display.update();
  }
}
```

Fig. 5. An implementation of the predicted control flow pointcut in SCoPE. (Simplified from the actual implementation, which uses a queue instead of recursively calling `pcflowGet`, and a set of checked methods for preventing stack overflow.)

```
aspect Memoization{
  pointcut memoPoint(int key): execution(int *(int)) && args(key)
                               && if(isSideEffectFree(thisJoinPoint));
  static boolean isSideEffectFree(JoinPoint jp) {
    for (Instruction i: traverseProceed(jp.getStaticPart()))
      if (i is a field set or field get)
        return false;
    return true;
  }
  ...the same field and advice declaration go here...
}
```

Fig. 6. An implementation of `Memoization` aspect in SCoPE.

Below, we briefly discuss how SCoPE copes with the issues of aspect interactions.

4.1 Double Compilation for Aspect Visibility

In order to make effects of aspects visible from the pointcuts, SCoPE takes the *double compilation* approach. When SCoPE compiles a program, it first generates an executable program in which all effects of aspects are woven into base classes yet leaving all `if` pointcuts as dynamic tests. It then evaluates the conditions in the `if` pointcuts with respect to each join point shadow and removes dynamic tests from the executable program. Aspect visibility is achieved by letting the `if` pointcuts inspect the woven program.

Our approach avoids paradoxical situations caused by self-observing aspects in contrast with other languages that support user-defined expressive pointcuts.

First, we explain why existing languages can cause paradoxical situations. Assume there is an expressive pointcut in an advice declaration that observes

the effects of the advice itself, and a language decides whether to weave the advice based on the result of the expressive pointcut with respect to a program before weaving aspects. If the result is true, it weaves the advice into the program, which might change the result of the expressive pointcut to false; i.e., the advice should not be woven into the program, which contradicts to the previous decision.

We believe the paradox is caused by the strategy to let the expressive pointcuts analyze the programs either in which the advice is *unconditionally* woven, or in which the advice is not woven at all. Hence the pointcuts can return different results.

We avoid this paradoxical situations by letting the expressive pointcuts analyze the program that *conditionally* executes the advice. Since our compiler merely optimizes the conditional branches at the second compilation, it is safe to reuse the result of the analysis with respect to the program before optimization.

4.2 Delimited Join Point Shadows

In order to appropriately delimit the target of analysis, we are designing an API for accessing bytecode instructions. For example, `traverseProceed` in Fig. 6 gives a sequence of bytecode instructions that will be executed during evaluation of `proceed` in an advice body. We also offer methods for traversing bytecode including the advice body itself, and including other preceding advice bodies. The design and implementation of the API are still in progress.

5 Related Work

There are several AOP languages that support user-defined expressive pointcuts [2–5]. We believe that practical programming experiences in those languages will highlight the aspect interaction issues discussed in the paper. However, as far as the authors understood, few of those explicitly addresses the aspect interaction with expressive pointcuts so far.

6 Conclusion

This paper presented how expressive pointcuts based on static analysis complicate aspect interactions due to the fact that the target of the analysis can also be affected by aspects and their precedence. It also showed the two required properties of expressive pointcuts, namely the aspect visibility and the delimited target of analysis, for enabling aspects interact correctly.

Our solution in the SCoPE compiler is to let expressive pointcuts analyze the woven program and to provide the high-level API for delimiting target of analysis. We implemented the SCoPE compiler[1] except for the high-level API.

[1] Publicly available at http://www.graco.c.u-tokyo.ac.jp/ppp/projects/scope/ .

References

1. Kiczales, G.: The fun has just begun. Keynote Speech at AOSD'03 (2003)
2. Chiba, S., Nakagawa, K.: Josh: an open AspectJ-like language. In Proceedings of AOSD'04 (2004) 102–111
3. Rho, T., Kniesel, G.: Uniform genericity for aspect languages. Technical Report IAI-TR-2004-4, Computer Science Department III, University of Bonn (2004)
4. Wu, P., Lieberherr, K.J.: Shadow programming: Reasoning about programs using lexical join point information. In Proceedings of GPCE'05. LNCS 3676 (2005)
5. Ostermann, K., Mezini, M., Bockisch, C.: Expressive pointcuts for increased modularity. In Proceedings of ECOOP 2005. LNCS 3586 (2005)
6. Burke, B., Brok, A.: Aspect-oriented programming and JBoss. Published on The O'Reilly Network (2003)
7. Aotani, T., Masuhara, H.: Compiling conditional pointcuts for user-level semantic pointcuts. In Proceedings SPLAT05 Workshop at AOSD'05 (2005) Online proceedings are available at http://www.daimi.au.dk/~eernst/splat05/.
8. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In Proceedings of ECOOP'01. LNCS 2072 (2001) 327–353
9. Walker, R.J., Murphy, G.C.: Implicit context: easing software evolution and reuse. SIGSOFT Softw. Eng. Notes **25**(6) (2000) 69–78
10. Walker, R.J., Viggers, K.: Implementing protocols via declarative event patterns. In Proceedings of FSE'04 (2004) 159–169
11. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In Proceedings of OOPSLA'05 (2005) 345–364
12. Sakurai, K., Masuhara, H., Ubayashi, N., Matsuura, S., Komiya, S.: Association aspects. In: Proceedings of AOSD'04 (2004) 16–25
13. Masuhara, H., Kawauchi, K.: Dataflow pointcut in aspect-oriented programming. In: Proceedings of APLAS'03 (2003) 105–121
14. Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: abc: an extensible AspectJ compiler. In Proceedings of AOSD'05 (2005) 87–98
15. Havinga, W., Nagy, I., Bergmans, L., Aksit, M.: Detecting and resolving ambiguities caused by inter-dependent introductions. In Proceedings of AOSD'06 (2006) 214–225
16. Klose, K., Ostermann, K.: Back to the future: Pointcuts as predicates over traces. In Proceedings of FOAL Workshop at AOSD'05 (2005) Online proceedings are available at http://www.cs.iastate.edu/~ leavens/FOAL/papers-2005/proceedings.pdf