

# On the Assessment of Pointcut Design in Evolving Aspect-Oriented Software\*

Raffi Khatchadourian<sup>†</sup>

Ohio State University  
khatchad@cse.ohio-state.edu

Phil Greenwood    Awais Rashid

Lancaster University  
{greenwop,awais}@comp.lancs.ac.uk

## 1. Introduction

The pointcut expression (PCE) is a key mechanism in enabling Aspect-Oriented Programming (AOP) (Kiczales et al. 1997) to improve the localization of crosscutting concerns. PCEs quantify over well-defined points in the execution of the program called *join points*. A *join point shadow*, on the other hand, refers to base-code corresponding to a join point (Xu and Rountev 2008), i.e., a point in the code where the compiler may perform the weaving (Masuhara et al. 2003). Advice *joins* at these points to allow the crosscutting concerns to be composed in an appropriate manner. PCEs need to be well-designed to ensure that they are correct in terms of identifying relevant join points to make certain the crosscutting concerns are composed correctly. Furthermore, PCEs should be robust in the midsts of base-code alterations. That is, changes to the base-code can lead to join points incorrectly falling in our out of scope of the pointcut expressions. Such situations are problematic in that they can cause crosscutting concerns to be composed incorrectly. If undetected, this could cause the composed program to behave unexpectedly, thus causing errors to occur. PCEs that result in such unexpected behavior of the composed program due to evolution are often referred to as “fragile” (Koppen and Stoerzer 2004).

The skill required to design a robust PCE, especially in languages such as AspectJ (Kiczales et al. 2001), is often considered a “dark-art”, as well as associated with many common pitfalls (Colyer et al. 2004). Typically, a number of alternative PCEs exist that are equivalent in terms of their

```
1 public class FooBar {
2     private int foo, bar;
3     public void setFoo(int f){this.foo = f;}
4     public void setBar(int b){this.bar = b;}
5 }
```

Figure 1. Example base-code.

end composition effect. For example, if all method executions within a class called `Test` are intended to be advised, multiple strategies may be employed. For instance, a generic PCE could be used that quantifies over all method executions (e.g., `execution(* Test.*(..)`), or each method could be enumerated individually (e.g., `execution(* Test.methodA(..)|| execution,...`).

Deciding which strategy is best in order to balance robustness, correctness, and precision is a non-trivial task. Apart from simple aforementioned PCEs, it is often impossible to ascertain prior to making maintenance changes whether the PCE will be, in fact, robust. Normally, it is only when maintenance changes have been made that fragile pointcuts are uncovered, which is an undesirable scenario. This paper outlines our intent to provide quantitative indicators in estimating the ability of a given PCE to preserve its semantics despite base-code alterations that may take place in the future. These indicators may then serve as a basis for suggesting alternative, more suitable PCEs.

## 2. Motivation

Consider the base-code snippet depicted in Figure 1 which defines a simple class `FooBar`. The class declares two integer fields, `foo` and `bar`, which are modified by two instance methods `setFoo(int)` and `setBar(int)`, respectively.

Suppose the developer wishes to advise the executions of methods that modify the state of `FooBar`. The most obvious PCE to capture such join points would take advantage of the `set` naming convention, possibly taking the form `execution(* FooBar.set*(..)`. Further suppose that the class is modified to introduce a method `incFoo()`, whose sole functionality is to increment the current value of `foo` by 1. Due to its construction, the current PCE would not capture

\* This material is based upon work supported by European Commission grants IST-33710 (AMPLE) and IST-2-004349 (AOSD-Europe).

<sup>†</sup> This work was administered during this author’s visit to the Computing Department, Lancaster University, United Kingdom.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Assessment of Contemporary Modularization Techniques 19<sup>th</sup> October 2008, Nashville, Tennessee, USA.

Copyright © 2008 ACM [to be supplied]...\$5.00

this new method; thus, the PCE, in this case, fails to capture the *true* intentions of the developer.

Through analyzing the currently advised shadows, we can extract a set of patterns that describe the underlying intentions of the developer. In this example, a common pattern exists that revolves around both advised shadows setting some field in `FooBar`. This pattern then can be subsequently applied to the modified version of `FooBar` and will suggest the newly introduced `incFoo()` method to be included in the PCE due to it also setting some field in `FooBar`. It is inevitable that patterns will be extracted which do not represent the developer's intentions and so cause incorrect suggestions to be made. To indicate the level of confidence in a pattern/suggestion quantitative indicators should be attached to each pattern, and subsequently each suggestion, to indicate how useful the suggestions may be. Such indicators can then be used to infer the how closely the original PCE captures the developer's intentions.

### 3. Pattern Metrics

The quality of the patterns can be measured in terms of the number of current shadows which they are representative of which can be used to infer their potential ability to capture new shadows in future versions of the software. However, it is equally important to measure the number of execution points which the pattern is also representative of but are not a shadow according to the PCE. This accuracy can be measured in terms of four indicators:

- True Positives (TP) - the number of actual shadows which the pattern matches.
- False Positives (FP) - execution points that match the pattern but are not a shadow.
- False Negatives (FN) - the number of actual shadows that are not matched by a particular pattern.
- True Negatives (TN) - counts how many potential shadows the pattern could have suggested but correctly did not.

From these four indicators a confidence metric can be calculated which is a ratio between recall and fall-out metrics:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

$$\text{Fall-Out} = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}}$$

$$\text{Confidence} = 1 - \frac{\text{Fall-Out}}{\text{Recall}}$$

A confidence value can be calculated for each PCE specified which is the average of all patterns derived from each PCE. This can be used to indicate how representative a particular PCE is of the underlying intentions of the developer

and subsequently how accurate it will be as changes are made to the base-code in terms of preventing shadows incorrectly falling in or out of scope.

Although the final confidence value for each PCE is useful in itself, the confidence values of each individual pattern that has been derived from the analysed PCE can be used to improve the design of the pointcut. For example, if an intention pattern is found with a high confidence (i.e. tending towards 100%) then the developer should look to express the pointcut in terms of that pattern. This is exemplified in the `FooBar` class whereby a set pattern is discovered which closer to the true intentions of the developer and is also able to ensure the newly introduced shadow is correctly advised.

### 4. Conclusion and Future Work

We have discussed initial insight into quantitatively assessing the quality of pointcut expressions in terms of their ability to accurately capture the underlying intentions of the developer. We envision a tool that would be able to predict the robustness of a given pointcut expression, thus reducing the need for pointcut maintenance. Future work consists of a rigorous treatment of the evaluation metrics, as well as an empirical evaluation of a tool possibly extending current approaches (Dagenais et al. 2007; Khatchadourian and Rashid 2008).

### References

- Adrian Colyer, Andy Clement, George Harley, and Matthew Webster. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. Addison-Wesley Professional, 2004.
- Barthélemy Dagenais, Silvia Breu, Frédéric Weigand Warr, and Martin P. Robillard. Inferring structural patterns for concern traceability in evolving software. In *Int. Conf. Automated Software Engineering*, 2007.
- Raffi Khatchadourian and Awais Rashid. Rejuvenate pointcut: A tool for pointcut expression recovery in evolving AO software. In *Int. Work. Conf. on Source Code Ana. and Manip.*, 2008.
- Gregor Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect oriented programming. In *Eur. Conf. Object-Oriented Programming*, 1997.
- Gregor Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. In *Eur. Conf. Object-Oriented Programming*, 2001.
- C. Koppen and M. Stoerzer. PCDiff: Attacking the fragile pointcut problem. In *Eur. Int. Workshop on Aspects in Software*, 2004.
- H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *International Conference on Compiler Construction*, 2003.
- Guoqing Xu and Atanas Rountev. Ajana: a general framework for source-code-level interprocedural dataflow analysis of aspectj software. In *Int. Conf. Aspect-Oriented Software Development*, 2008.