

Towards Tool-supported Update of Pointcuts in AO Refactoring

Jan Wloka*
Fraunhofer FIRST
Berlin, Germany
jan.wloka@first.fraunhofer.de

Abstract

Aspect-oriented programming (AOP) is often introduced as an extension to a programming language. The new modularization mechanisms are provided by new language constructs, such as the pointcut and the advice. Pointcuts specify where and when an advice is executed and thereby refer to other program elements and structures to express the execution conditions. During the evolution of a program these referenced structures might be changed and hence the advice is not invoked as intended.

In this paper we present an approach for assessing the impact of source code changes on pointcuts and a program analysis that supports the identification of broken pointcuts. We elaborate how a refactoring tool can determine reasons and how an equivalent pointcut update can be calculated.

1 Introduction

Refactoring was found to improve the design of an existing software system in a safe and reliable way. It allows to modify a program's structure while it preserves its observable behavior. Especially tool-supported refactoring has become very important in software engineering for controlled software evolution. While refactoring tools for object-oriented programs are commonly used, it is not fully explored yet how the effects of even local changes on the behavior of aspect-oriented programs can be determined.

AOP introduces new modularization mechanisms

*This work has been supported by the German Federal Ministry for Education and Research under the grant 01ISC04A (Project TOPPrax).

to encapsulate implementations that would otherwise be spread over multiple modules. An *aspect*, as new implementation module, allows for example to adapt the behavior of other modules. Therefore, it provides new language constructs, mainly pointcut and advice, to specify where and how the program behavior should be adapted. A *pointcut* binds an advice to well-defined points in the program execution. These so called *joinpoints* are selected by specifying their properties. E.g., method call joinpoints can be selected by specifying their method's signature. Pointcuts refer to various information of a program to express a joinpoint property. An *advice* implements the additional behavior and are executed before, after or around a selected joinpoint. If a refactoring tool should apply a structural improvement in a behavior preserving way, it must be able to determine the change impact on all pointcuts in an AO program. Otherwise, the program behavior cannot be preserved.

In this paper we present a program analysis technique for determining the impact of source code changes on pointcuts in aspect-oriented programs. We illustrate how pointcut matches can be represented to determine the change impact and discuss several factors that influence the analysis results. Based on this analysis we describe different situations for updating a pointcut and illustrate with an example how the proposed program analysis works. The remainder of the paper is structured as follows. In section 2 the term joinpoint property is defined as the atomic part of a pointcut and examples are given. Section 3 presents some example code that is used in the following sections and applies one refactoring on it. Section 4 illustrates our approach for assessing the change impact on pointcuts, followed by a brief discussion in section 5 when and how an equivalent

update for a pointcut can be calculated. In section 6 this paper ends with some implementation details and a preliminary conclusion.

2 Specifying Joinpoint Properties

A pointcut can refer to very different kinds of program information. In general one distinguishes between static and dynamic pointcuts. A *static pointcut* uses only information that can be obtained from the program code, whereas a *dynamic pointcut* also employs runtime information.

In both cases, a pointcut specifies structural properties of a program representation that is either built from the program code (e.g., an abstract syntax tree, AST), or from runtime information during its execution (such as a stack trace). Therefore we define a joinpoint property as follows:

DEFINITION: A *joinpoint property* is a structural property of a program representation that can be built from static or dynamic program information.

Pointcuts specify different kinds of joinpoint properties and thus refer to different information of a program. In recent publications [6, 1, 10, 2] various joinpoint properties, like naming, containment, inheritance relationships, method execution order and instance reachability in object-graphs, have been proposed for joinpoint selection in pointcuts.

From a software evolution point of view all these properties can be altered by different changes in a program's source. Two limitations need to be considered if the change impact on every property should be determined: how specific a property is defined within a pointcut and how good it can be approximated during evaluation.

Name patterns which are provided by most AOP approaches, allow a very fuzzy specification. Method executions of certain methods can be selected using AspectJ syntax[11] by expressing only some parts of the signature, like `execution(* *.foo(..)`.

In this case only the method name can be used to decide whether additional joinpoints are intended.

More recent approaches, e.g., stateful aspects in JAsCo[10] or Tracematches[1], allow to specify more "semantic" joinpoint properties¹. They allow to spec-

¹The term *semantic* is used here in the context of program execution, to indicate that a joinpoint property associated with the behavior that the program exhibits.

ify a certain execution sequence as joinpoints, i.e., whenever this sequence is executed an associated advice will be invoked. The use of an arbitrary execution sequence causes another kind of problems. Situations in which a certain sequence appears can be approximated only in some cases and the reasons for an altered situations can rarely be determined from the static structure of a program.

3 An Example "Push Down Method"

In the following, we will be using the source code example from Listing 1, which is implemented in AspectJ[3].

Listing 1: Code Example

```
1 package p1;
2 public aspect A {
3     pointcut posChanged(): set(int *);
4     before(): posChanged() {
5         System.out.println("Changing_
6             position");
7     }
8 }
9 package p1;
10 public class B {
11     int pos;
12     static void main(String[] args) {
13         C c = new C();
14         c.setPos(1);
15         c.update();
16     }
17     void setPos(int pos) {
18         this.pos = pos;
19     }
20 // will be moved during the refactoring
21     void update() {
22         pos = pos + 10;
23     }
24 }
25
26 package p1;
27 class C extends B {
28 }
```

The `main` method in `class B` creates a new instance of `class C`, sets the value of field `pos` to 1 by invoking `C.setPos(int)` and calls the method `C.update()` to increase the field `pos` by 10.

The `aspect A` intercepts the execution of any assignment to field `pos` and prints some status information to the console before the field `pos` is modified.

In a small experiment we now modify the code using the standard Java "Push Down Method" refac-

toring pattern (see [5] page 328). The refactoring is applied to method `B.update()` in order to move it to the subclass `C`.

In the following we present how the standard Java refactoring pattern influences the pointcut of aspect `A`, how this can be determined by a refactoring tool, and how a tool can calculate whether the pointcut needs to be updated.

4 Assessing the Change Impact on Pointcuts

A pointcut binds a set of joinpoints to an advice in order to have the advice executed at every bound joinpoint. Stoerzer and Graf have classified code changes that influence the program behavior defined by such advice bindings. In [9] they distinguish between three different kinds of changes:

- change of joinpoint property specifications in pointcuts
- change of implementation of the bound advice
- change of the base code which exhibits the addressed joinpoints

For brevity and because its the most difficult one, we focus in this paper on base code changes.

Representing base code changes. Very different changes can be applied to a software system. They differ in extend and complexity, and can range from a local text edit with no further impact, to huge adaptations that affects multiple implementation modules. Similar to Ryder et al. in [8, 7], we have developed an abstract change representation that consists of atomic changes.

These *atomic changes* represents modifications to program elements at any level of detail, such as type, field and expression. Smaller changes either can be ignored, or are represented as a property of a specific program element. More complex base code changes are then composed of atomic changes. This leads to several advantages and simplifies the change analysis. If we consider the *Push Down Method Refactoring* in our example code the change representation would look like the change tree shown in Figure 1. The tree contains every changed program element as well as its parents. Each element is associated to an atomic change that stores the change reason. For the example, the tree in the figure shows the removal of

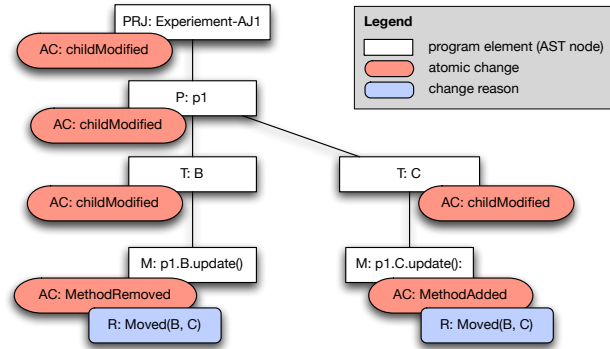


Figure 1: Atomic Change Tree representing the change for *Push Down Method*

method `update()` in class `B` and the addition of the method in class `C`. The associated reason indicates that both changes are caused by this method move.

Calculate a pointcut’s selection. A pointcut specifies joinpoint properties to bind an advice to a set of joinpoints, i.e., an associated advice is bound to every *pointcut match*. A pointcut, however, may also refer to other program elements that are used to express a certain joinpoint property. For example, the pointcut `call(public void p1.B.update())` would refer to the fully qualified name of class `B`, even if the actual joinpoint may be located somewhere else. For the specification of every joinpoint property a pointcut selects other program elements. In the following, we distinguish between *partial pointcut matches* (only a subset of the joinpoint properties) and *complete pointcut matches* (all joinpoint properties). Based on this distinction we define a pointcut selection as follows:

DEFINITION: A *pointcut selection* contains the mapping of partial or complete matches to the matched program elements as well as all program elements referenced by the pointcut.

In other words, a pointcut selection holds every information of a certain program that is needed to determine the pointcut’s matches, including every partial match.

Figure 2 shows the pointcut selection for `set(int *)` of our example program. It holds all program elements that are directly referenced by the pointcut, including their parent nodes, before the refactoring is applied.

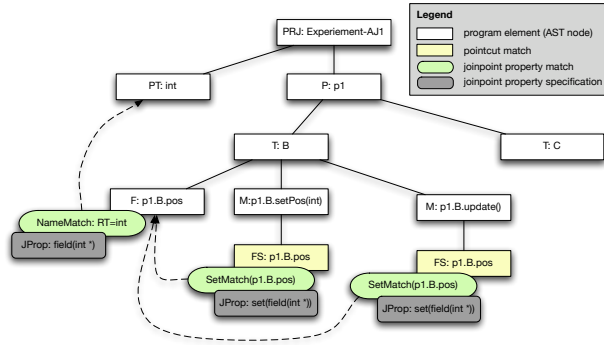


Figure 2: Pointcut Selection for `pointcut set(int *)` before the change

In our example the pointcut would match at the field assignments in `setPos(int)` and `update()` of class `B`, as shown in the selection tree. Additionally both `SetMatch` matches refer to the field `pos` with one `NameMatch` match which in turn references the primitive type `int`.

Determine the change impact. The *Push Down Method Refactoring* is virtually applied to get the change information. For the modified program a second pointcut selection is calculated for every pointcut in the system. Both versions are compared and a *pointcut selection delta* is produced. This delta contains all partial and complete pointcut matches that aren't equal in both versions. In this way the impact of a source code change can be represented in terms of a pointcut.

For our code example the pointcut selection delta is shown in Figure 3. It contains two matches (one added and one removed) as well as all enclosing and referenced elements. The `SetMatch` matches at both field sets `FS` of `pos` are indicated as modified, because the matches are identified by a fully qualified name².

5 Propose a Pointcut Update

The overall goal in a refactoring process is to keep the set of selected joinpoints semantically equal. In particular, this means a pointcut is updated in a way that it matches the same program elements as before³. In the following we briefly discuss different

²Statement level pointcut matches may have an ambiguous fully qualified name. For a unique identification their number of appearance in the enclosing block is added to the name.

³Since pointcut matching is done on program representations, a pointcut can match either at a program element itself

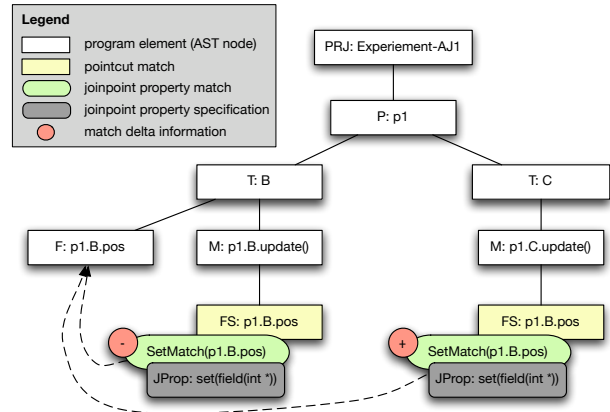


Figure 3: Pointcut Selection Delta for `pointcut set(int *)`

update situations.

No update required. We've identified three situations in which a pointcut might be affected but doesn't need an update.

- *Unaffected pointcut:* The pointcut selection delta is empty, i.e., all program elements referenced by this pointcut are not affected by the change.
- *Unaffected pointcut matches:* The pointcut selection delta is not empty, but no complete pointcut match can be found in the delta. I.e., all matches in the delta are partial and thus not relevant for the advice execution. In such cases the pointcut refers to affected program elements but the resulting set of pointcut matches is not altered.
- *No affected joinpoint property specification:* The delta contains added and removed (complete) pointcut matches. However, for every removed pointcut match an equivalent added match can be found. This seems often the case when some program elements were moved around, but its actual location isn't specified in the pointcut.

Update required. If some complete pointcut matches have been added or removed, an update of the actual pointcut is necessary in order to ensure behavior preservation. Updating a pointcut means updating every specification of an added or removed (static) or at the execution of the program element (dynamic).

joinpoint property match. For every altered pointcut match, the referenced joinpoint property matches are analyzed. If an equivalent joinpoint property specification can be calculated, an update is proposed. Otherwise the update is unresolvable. Both these cases as well as relevant causes are briefly discussed in the following:

- *Resolvable Update:* If the affected joinpoint property specification can be directly mapped to a base code change, then the refactoring together with this change are sufficient to derive an updated joinpoint specification. Interestingly, this often seems to be the case when program elements are renamed, extracted, inlined or moved.

- *Unresolvable Update:* Until now we have some possible reasons identified in which our approach cannot provide sufficient impact information.

Joinpoint property removal: A source code change removes a program element or a structure that is directly referenced by a pointcut. A new joinpoint property needs to be found to select the joinpoint.

Fuzzy joinpoint property meaning: Some pointcuts specify only very few concrete information, e.g., together with name patterns. A name pattern can partially specify the name of a program element, such as `call(* * get*(..))`. If an additional match with such a name is detected a tool has only the three letters 'set' to decide whether the new match is valid.

Insufficient approximation of dynamic pointcut matches: Our approach is based on static program analysis. Therefore, all dynamic joinpoint properties needs to be approximated, i.e., actually the employed program representation. For example, used of runtime values such as in `if()` pointcuts cannot be approximated at all.

Modified aspect interaction: If the proposed pointcut update would refer to a joinpoint that is also selected by another pointcut, a tool couldn't decide in which order the advices should be performed.

In such situations the developer at least should be provided with sufficiently detailed information about identified conflicts or limitations.

Pointcut update calculation. Figure 4 presents the complete change impact tree for the refactoring in our example. In this case all affected pointcut matches can be directly mapped to a base code

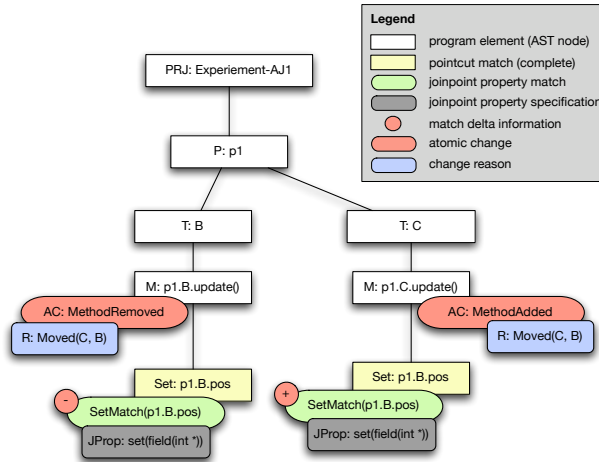


Figure 4: Change impact representation for pointcut `set(int *)`

change of an enclosing element, i.e., no pointcut match (not even a partial match) is affected by the refactoring. In this case the pointcut would be left unchanged, because no program information referenced by a pointcut is affected.

6 Conclusions and Future Work

We have shown how static program analysis can be used to determine the change impact on pointcuts and how the impact can be represented in order calculate an adequate update proposal within a refactoring process. We have briefly discussed different factors that influence the calculation of a pointcut update and illustrated our approach with an example.

The program analysis approach presented in this paper has been implemented by a small analysis framework called "Soothsayer". The implementation is part of the ObjectTeams/Java IDE[4] and based on Eclipse data structures. In previous work some Java refactoring patterns have been adapted for ObjectTeams/Java that were not concerned about pointcuts. The Soothsayer framework aims to extend this implementation in order to cope with pointcuts during a refactoring's application. Currently only of few kinds of joinpoint properties are supported, like name patterns, containment and inheritance relationships.

First experiences have shown that our analysis is sufficiently represents the impact information, so in several situations an adequate pointcut update could be proposed. In the near future we are going to test our program analysis with more kinds of joinpoint properties and improve the calculation of pointcut updates, to consider more situations for resolvable updates. Soothsayer isn't currently really integrated with the refactoring tool, so the analysis results cannot be used to apply the actual pointcut updates. We're also going to do that.

Acknowledgements

Thanks to Thomas Dudziak for providing valuable comments on an earlier draft of the paper.

References

- [1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhotk, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching to AspectJ. Technical Report abc-2005-1, Programming Tools Group, University of Oxford University, UK; BRICS, Group of Aarhus, Denmark; Sable Research, McGill University, Montreal, Canada, 2005.
- [2] Michael Eichberg, Mira Mezini, and Klaus Ostermann. Pointcuts as functional queries. In Wei-Ngan Chin, editor, *Programming Languages and Systems: Second Asian Symposium, APLAS 2004*, Lecture Notes in Computer Science, pages 366–382, Taipei, Taiwan, November 2004. Springer-Verlag Heidelberg.
- [3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.
- [4] Object Teams home page. <http://www.ObjectTeams.org>.
- [5] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [6] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In Olaf Spinczyk, Andreas Gal, and Michael Schoettner, editors, *ECOOP Workshop on Programming Languages and Operating Systems*, 2004.
- [7] Xiaoxia Ren, Barbara G. Ryder, Maximilian Stoerzer, and Frank Tip. Chianti: a change impact analysis tool for java programs. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 664–665, New York, NY, USA, 2005. ACM Press.
- [8] Barbara G. Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53, New York, NY, USA, 2001. ACM Press.
- [9] Maximilian Störzer and Jürgen Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM*, pages 653–656. IEEE Computer Society, 2005.
- [10] System and Software Engineering lab (SSEL) at the Department of (Applied) Computer Science (Faculty of Sciences) at Vrije Universiteit Brussel (VUB). *JAsCo Language Reference*. available from <http://ssel.vub.ac.be/jasco/documentation:main>.
- [11] Xerox Corporation. *AspectJ Programming Guide*. available from <http://eclipse.org/aspectj>.