Inductively Generated Pointcuts to Support Refactoring to Aspects

Tom Tourwé
Centrum voor Wiskunde en Informatica
P.O. Box 94079, NL-1090 GB Amsterdam
The Netherlands
Email: tom.tourwe@cwi.nl

Andy Kellens
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
Email: akellens@vub.ac.be

Wim Vanderperren & Frederik Vannieuwenhuyse
System and Software Engineering Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
Email: { wvdperre, frvnieuw }@vub.ac.be

Abstract

In this paper, we show that the basic pointcut languages offered by current aspect-oriented programming languages impact important software engineering properties, such as evolvability and comprehensibility, in a negative way. We discuss this impact by means of detailed examples, and propose an advanced pointcut managing environment, based on machine learning techniques, to overcome the problem.

1 Problem Statement

An important part of an aspect's definition consists of the specification of its pointcuts, e.g. those places in the source code where the aspect has to exert its influence. Current-day aspect-oriented programming languages, such as as AspectJ [4], only incorporate a very basic pointcut language, however, which forces pointcuts to rely heavily on the existing structure of a software application. Moreover, the way in which these pointcuts need to be specified is particularly primitive: all responsibility is left to the developer, who not only needs to identify the pointcuts, but also has to make sure they are specified in the correct way. Consequently, it is notoriously hard to define generic, reusable and comprehensible pointcuts that are not tightly coupled to an application's structure.

The success and usefulness of the aspect-oriented software development approach raises the important issue of how an object-oriented application can be refactored into an aspect-oriented one. The observation presented above is especially important in that context:

- the basic pointcut language may prohibit us from capturing the application's structure adequately, and may thus not allow us to define the pointcuts in the intended and most generic way possible. To do so, the structure first needs to be refactored;
- although aspect-specific refactorings are available that extract aspects from the classes and methods of an object-oriented application, they rely on the developer to determine the pointcuts of the extracted aspects and define them in the right way. As such, it is impossible to guarantee the correctness of these pointcuts, and thus the behaviour preservation of the refactoring;
- due to the tight coupling of the pointcuts to the application's structure, the pointcuts of the extracted aspects are affected by subsequent refactorings of the application. Although this impact has been studied already [3], proposed solutions to the problem merely "patch" the affected pointcuts. instead of radically reconsidering and redefining them appropriately. The pointcuts themselves will thus gradually degenerate after a small number of refactorings, decreasing their comprehensibility.

In the next paragraphs, we provide an illustrative example of all three problems.

1.1 Pointcut Definition

Consider the following two classes `A` and `B`, that are not related by inheritance, and their respective methods `m` and `n`:

```
class A {
   public void m() {
      this.c();
      x.a();
      y.b();
   }}

class B {
   public void n() {
      x.a();
      y.b();
      this.d();
   }}
```

We now want to insert some specific code dealing with a non-functional aspect A around the calls to methods `a` and `b` as follows:

```
class A {
   public void m() {
      this.c();
      <insert aspect X's before code here>
      x.a();
      y.b();
      <insert aspect X's after code here>
   }}

class B {
   public void n() {
      <insert aspect X's before code here>
      x.a();
      y.b();
      <insert aspect X's after code here>
      this.d();
   }}
```

The basic pointcut language offered by AspectJ does not allow us to define this aspect in a straightforward way. Indeed, the language can not express the pointcut we need here: "all calls to method `a` followed by a call to method `b`". We can specify the aspect as follows, using two separate pointcuts for each method call and two separate advices to insert the code before and after the respective pointcut:

```
aspect X {
    pointcut pc1(): call(* *.a());
    pointcut pc2(): call(* *.b());
    before(): pc1() {
       <aspect X's before code here>
    }
    after(): pc2() {
       <aspect X's after code here>
    }}
```

This is however not guaranteed to be correct at all times, since the aspect will also work on code where the order in which `a` and `b` are called is different, or on code where only `a` or `b` is called.

The only option left to define the pointcut in the correct way is to refactor the classes `A` and `B` first, as follows:

```
class A {
   private void ab() {
      x.a();
      y.b(); }
   public void m() {
      this.c();
      this.ab();
   }}

class B {
   private void ab() {
      x.a();
      y.b(); }
   public void n() {
      this.ab();
      this.d();
   }}
```

Note that the code in the `ab` method can not be factored out, because classes `A` and `B` are not related by inheritance.

This refactoring makes the aspect's definition more clear and easier to understand:

```
aspect X {
    pointcut pc(): call(* *.ab());
    around(): pc() {
       <aspect X's before code here>
       proceed();
       <aspect X's after code here>
    }}
```

The fact that we need to perform a refactoring before we can define a correct aspect, is quite cumbersome. First of all, we want the base code to be oblivious from the aspects that are applied to it, which is what aspect-oriented programming is all about. Second, refactorings are traditionally applied to make the code cleaner from a conceptual point of view. In this case, the code is refactored to be able to introduce the aspect, and it is thus not guaranteed that it becomes cleaner. Third, the solution is not scalable when multiple aspects should be introduced onto the same code. Each aspect may require a different refactoring, and different refactorings may conflict with one another, because they try to refactor the same code in different ways.

1.2 Pointcut Extraction

Consider the following code:

```
class A {
   private void c() { ... }
   public void m() {
      x.a();
      this.c();
      y.b();
   }}
class B {
   private void d() { ... }
   public void n() {
      x.a();
      this.d();
      y.b();
   }}
```

where the method calls to `a` and `b` this time represent a non-functional concern, such as transaction management, error handling, etc. This concern can be captured more cleanly in an aspect, of course. So we apply an *Extract Advice* refactoring [3] to get the following code:

```
class A {
   private void c() { ... }
   public void m() {
      this.c();
   }}
```

```
class B {
   private void d() { ... }
   public void n() {
      this.d();
  }}
aspect X {
   pointcut pc() : ...
   around() : pc() {
      x.a();
      proceed();
      y.b();
   }}
```

The hardest part of this particular refactoring is determining the pointcut of the `X` aspect. Using the restricted pointcut language of AspectJ, we can choose among the following alternatives:

- The pointcut describes the execution of the methods `m` and `n` in classes `A` and `B`: `execution(void A.m()) || execution(void B.n())`. If there are no other classes implementing either `m` or `n`, the pointcut can be made more generic by using wildcards: `execution(* *.m()) || execution(* *.n())`. This may prove beneficial if later on, other `m` or `n` methods are defined;
- The pointcut describes calls to methods `c` and `d` around which the aspect has to be executed: `call(void A.c()) || call(void B.d())`. If besides `m` and `n`, no other methods in the application call `c` or `d`, the pointcut can once again be made more generic by using wildcards: `call(* *.c()) || call(* *.d())`;

Clearly, whichever option we choose, it will always be tightly coupled with the application's structure, since the pointcuts explicitly include a method's signature. Consequently, when the application evolves, it is very likely that the pointcut needs to be changed as well, which is quite cumbersome. We believe the root cause of this problem to be the fact that a developer is forced to provide a detailed definition of a pointcut when defining an aspect, and that only a method's signature can be taken into account for this purpose. Consequently, the developer can only choose between a limited number of alternatives, and he has no clue which is the best one, since he does not know how the application will evolve.

1.3 Pointcut Refactoring

Consider the following class and method definition and the aspect that works on them:

```
class A {
   public void m() {
      ...
   }}
aspect X {
   pointcut pc1() :
      call(void *.m());
   ...
   pointcut pc2() :
      call(void *.n());
   ... }
```

and suppose we apply a *Rename Method* refactoring to rename the method `m` in class `A` to `n`. Clearly, this impacts the two pointcuts `pc1` and `pc2`, since these deal with `m` and `n` methods respectively. As defined by [3], we have to change these pointcuts accordingly as follows, to ensure they remain correct:

```
class A {
   public void n() {
      ...
}}
aspect X {
   pointcut pc1():
      call(void *.m()) || call(void A.n());
   ...
   pointcut pc2():
      call(void *.n()) && !call(void A.n())
   ... }
```

Other refactorings, such as *Move Method* and *Extract Method* may change the pointcuts in other ways, always using a combination of logic operations as appropriate [3]. It is easy to see that such an approach will lead to pointcuts that become quite complex and hard to understand, let alone that the impact of the refactorings can be assessed in such an easy way as in this example. Ultimately, these pointcuts will need to be refactored as well.

Once again, the main problem here is that a developer has to deal with very detailed and specific pointcuts explicitly. The fact that these pointcuts become overly complex after a few refactorings, and that there is no support for reducing or managing this complexity, makes this a very difficult and error-prone task.

1.4 Summary

To summarise, the three problems we identify are:

- the pointcut language offered by current-day aspect-oriented programming languages is too primitive and not expressive enough;
- pointcuts are very tightly coupled to an application's structure;
- developers are forced to deal with pointcuts at too low a level.

In the remainder of this paper, we propose an advanced aspect-oriented programming environment that tackles these problems.

2 Description

As a solution to the three problems mentioned above, we propose to include the notion of *inductively generated pointcuts* in the aspect-oriented programming language. This includes both an advanced pointcut language and a supporting environment that manages the pointcuts for the developer. The idea is that very complex pointcuts can be expressed in the language, but that the developer is shielded from this complexity by the environment. In this way, the developer no longer has to manage the pointcuts in very close and specific detail, but simply defines a *pointcut classification*, that groups the source code entities that should be contained in a pointcut, and relies on the environment to define the pointcut automatically in the correct way. To this extent, the environment uses a machine learning algorithm that is able to derive an intentional definition of the pointcut.

2.1 Pointcut Language

Internally, the pointcut managing environment uses the logic language Soul [8] as a pointcut language. Soul is a dialect of Prolog, that is implemented on top of, and tightly connected to the Smalltalk programming environment. In other words, Soul is a Turing-complete programming language that can be used to reason about object-oriented applications in a concise and straightforward way. It has already been argued several times that such a language is extremely well-suited to express pointcuts [2], in particular because it offers full access to the source code, and thus allows pointcuts to use any kind of information necessary to adequately and precisely describe their points of interest. An extension to Soul, SoulJava [1], allows us to use the reasoning capabilities of Soul on Java code.

2.2 Pointcut Managing Environment

The environment that manages the pointcuts consists of two main parts: a graphical user interface that offers a view on the source code, and an inductive logic programming algorithm that is responsible for computing the pointcut definition. The source code browser allows the developer to define a *pointcut classification* that groups certain points in the source code that should be included in a particular pointcut. These points can be classes, methods, variables or individual statements. The inductive logic programming algorithm then uses the elements of these pointcut classifications as examples to compute the actual pointcut definition. It does this by considering the various properties that the provided examples share, such as the messages sent by a method, the methods that access a particular variable, or the methods in which a particular statement occurs. The result of the algorithm is a set of logic rules and/or logic facts in the Soul language, that compute the pointcut elements.

2.2.1 Specifying Pointcuts

Pointcuts are specified by the developer in a specialised graphical user interface, using a simple drag and drop approach. The user interface is an extension of the StarBrowser, and offers the developer a list of existing pointcut classifications as well as a view on the source code of the application. Facilities are provided to change existing pointcut classifications, by adding elements to or removing elements from them. A developer can also define a new pointcut classification, that initially contains no elements. Then, he/she can just drag and drop source code entities that should form part of the pointcut into this classification.

2.2.2 Inductive Logic Programming

Inductive logic programming (ILP) [6] is a machine-learning technique that, given a set of desired example solutions for a logic predicate, automatically induces rules for that predicate. To derive the rules, an already established set of predicates, called the *background knowledge*, is used. The technique aims to uncover some general pattern in the examples, so that additional examples with the same pattern will be covered by the rules as well.

In our environment, inductive logic programming is used to uncover a pattern between a number of source code entities that a developer classifies in an aspect. This pattern will then represent the pointcut of the aspect. The background knowledge used by the induction algorithm consists of all predicates that Soul offers to reason about source code. Among others, this knowledge contains predicates such as `classImplements`, `superclassOf`, `subclassOf`, `senderOf`, ...

The following example demonstrates this use of ILP:

```
class A extends C {
   private void c() {};
   public void m() {
      this.c();
   }}

class B extends C {
   private void d() {};
   public void m(){
      this.d();
   }}
```

The example consists out of classes `A` and `B` which are subclasses of `C`. Both classes implement a method `m` that performs a call to a private method, respectively the methods `c` and `d`. Suppose we want to induce a pointcut `pointcut1` which covers all exections of the method `m`. Table 1 gives an overview of the examples and background knowledge which are used in our example and shows the induced Soul rules [1]. As examples of the pointcut, the ILP

[1] The variables in the induced logic rules have automatically generated names, which we edited for readability.

Examples	Background knowledge	Induced Logic Rules
`pointcut1(method(A,m))`	`implementsMethod(A,m)`	`pointcut1(method(?class,?method)) if`
`pointcut1(method(B,m))`	`implementsMethod(A,c)`	`  implementsMethod(?class,m),`
	`subclass(A,C)`	`  subclass(?class,?C),`
	`performsPrivateSelfSend(A,m)`	`  performsPrivateSelfSend(?class,?method).`
	`implementsMethod(B,m)`	
	`implementsMethod(B,d)`	
	...	

Table 1. Examples, background theory and induced rules for the pointcut1 predicate

algorithm is given the methods `m` of classes `A` and `B`. The background knowledge is obtained by analyzing the examples using Soul and gathering the useful information. Notice that the induced rules express the examples in function of the background predicates.

In order to generate these rules, the ILP algorithm uses a bottom-up technique (although other techniques exist [5]) that generalises a set of specific examples into a more general rule. For example, given the two logic clauses `subclass(A,C)` and `subclass(B,C)`, the technique automatically deduces the clause `subclass(?x,C)`. Because each pair of logic clauses in the examples and the background knowledge is considered in this way, the technique is able to construct a set of logic rules that matches all examples. Since this set of rules contains a lot of redundant parts, a reduction is applied to obtain logic rules as shown above. A discussion of this entire technique, which is called *relative least general generalisation* is beyond the scope of this paper. We refer the interested reader to [7].

Note that the obtained rule is quite general, but can be adapted by the environment in the appropriate way as soon as the developer adds or removes elements from the classification of the aspect. In doing so, the developer actually provides positive and negative examples to the machine learner, which makes sure the rule that is derived after a number of iterations has high chances of corresponding exactly to the intention of the aspect.

2.3 Discussion

The inductive pointcut model discussed above is capable of relieving the problems we discussed above, as follows:

- since the pointcut language is more expressive, the need to refactor an application's structure in order to be able to capture it in a pointcut disappears;
- the developer does not need to consider various alternatives for defining a pointcut. since the pointcuts are automatically defined and managed by the environment. The environment considers the alternatives and generates a pointcut that includes the provided examples and does not include any other part of the program;
- the complexity introduced by refactorings that have an impact on a pointcut is no longer a problem, since the developer does not deal with the pointcuts explicitly. When a refactoring is applied, its impact on the existing pointcuts can be analysed automatically, and the inductive logic programming algorithm can recompute a new, appropriate pointcut if necessary. Moreover, the environment always defines a pointcut as simple as possible (as opposed to a complex combination of logic operations), and this makes it easier to assess the impact of a refactoring;

3 Do Inductively Generated Pointcuts Support Comprehensibility?

3.1 How do Inductively Generated Pointcuts Support Comprehensibility?

Inductively generated pointcuts support comprehensibility simply because the developer deals with the source code artifacts that make up the pointcut, as opposed to the abstract description of these artifacts. In this way, the pointcut managing environment offers the developer a comprehensive overview of all artifacts covered by a particular pointcut.

Moreover, the pointcut description that is generated by the inductive logic programming algorithm can be inspected, if necessary. Because the algorithm recomputes a pointcut whenever necessary (for example, because a refactoring was applied), this pointcut is always as simple as possible (as opposed to a complex combination of logic operations).

3.2 How do Inductively Generated Pointcuts Reduce Comprehensibility?

Since the inductive logic programming algorithm computes the pointcut definition automatically, the developer no longer has precise control over this definition. In some cases, this may be problematic. For example, it is very difficult to determine the cause of an incorrect pointcut definition or to debug it. This requires either removing source code artifacts from the pointcut classification, or adding

other ones, or inspecting the generated logic rule and trying to adapt it manually. On the other hand, one can easily imagine that debugging an incorrect AspectJ pointcut definition may be just as difficult.

4 Do Inductively Generated Pointcuts Support Evolvability?

4.1 How do Inductively Generated Pointcuts Support Evolvability?

Inductively generated pointcuts support evolvability in a number of ways.

First of all, since the pointcut language used is very expressive, the pointcut can be described in a more intentional way. In other words, the pointcut models the semantics of the source code artifacts, as opposed to just the structure. As such, inductively generated pointcuts reduce the tight coupling between the aspect and the source code, making the former more reusable, while the latter can evolve more easily without affecting existing pointcuts.

Second, inductively generated pointcuts help the developer when refactoring an object-oriented application into an aspect-oriented one. The environment relieves the developer from the difficult, error-prone and time-consuming task of identifying and defining the appropriate pointcuts for the extracted aspects. The developer only has to specify the source code artifacts to which a particular aspect should be applied, and the pointcut describing these artifacts is generated automatically.

Last, because the environment manages the pointcut descriptions automatically, the developer no longer has to worry about the impact of a particular refactoring of the base code on the existing pointcuts. The environment is able to assess this impact automatically, and adapt the pointcuts as necessary. In this way, we can be sure that the pointcut's definition is always as simple as possible (but still contain enough complexity to remain correct).

4.2 How do Inductively Generated Pointcuts Reduce Evolvability?

When a lot of new classes and methods are added to the application at the same time, these should all be classified into the correct pointcut classifications. This may be a time-consuming activity. A particularly attractive property of machine learning algorithms in this context is that they perform better over time, when more examples become available. As such, we believe that the pointcuts that are generated will become quite sophisticated and will come quite close to an intentional description, which makes them more robust toward evolution.

We should also point out that the situation is as problematic in AspectJ: when a lot of classes and methods are added to an application, the impact on existing pointcut definitions has to be considered as well. A developer may need to inspect each and every pointcut, in order to verify whether it does not include any of the new entities it shouldn't, or whether it does include all new entities it should.

5 Summary

In this paper, we show how pointcut definition, extraction and refactoring turns out to be problematic and impact important software engineering properties in a negative way. We argue that this was due to the fact that the pointcut language offered by current-day aspect-oriented programming languages is too primitive , that pointcuts are very tightly coupled to an application's structure, and that developers are forced to deal with pointcuts at too low a level.

To overcome these problems, we propose the notion of inductively generated pointcuts. These allow a developer to work with pointcuts at a high-level of abstraction, by merely moving them into the appropriate pointcut classification. The pointcut descriptions are automatically generated from these classifications, by means of an inductive logic programming algorithm. Moreover, the impact of particular changes to the base code can be assessed automatically, and the pointcuts will be adapted automatically when necessary.

References

[1] J. Fabry and T. Mens. Language-independent detection of object-oriented design patterns. *Elsevier Computer Languages*, To Be Published.

[2] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the Second International Conference of Aspect-Oriented Software Development*, 2003.

[3] S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of aspect-oriented software. In *Net. ObjectDays 2003, Erfurt, Germany*, 2003.

[4] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Grisworld. An overview of AspectJ.

[5] T. Mitchell. *Machine Learning*. McGraw-Hill International Editions, 1997.

[6] S. Muggleton. *Inductive logic programming*. London: Academic Press, 1992.

[7] S. Muggleton and C. Feng. Efficient induction of logic programs. In *First Conference on Algorithmic Learning Theory*, 1990.

[8] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.