An Experiment in Using Inductive Logic Programming to Uncover Pointcuts

Kris Gybels* and Andy Kellens†
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
Belgium
{kris.gybels, andy.kellens}@vub.ac.be

July 9, 2004

Abstract

The subject of this paper is the transformation from pre-AOP legacy software to aspect-oriented software. To factor out crosscutting concerns from such software we propose the use of techniques to automate the task of uncovering pointcuts. We discuss problems inherent in this task and propose the use of inductive reasoning techniques for the automation. We apply such a technique to one kind of aspects: unique methods which can be found in a standard Smalltalk image.

1 Introduction

Now that AOP has reached a certain level of maturity, there is a growing interest in applying it not only to new projects but to existing systems as well. Existing systems have been modularized using only classical techniques like procedures, functions, classes etc. which we know to be insufficient for modularizing certain concerns [5]. This complicates the job of the maintainer who is often not even one of the original developers. With the advent of AOP such concerns can now be properly modularized which further postpones the system's expiration date by improving its maintainability.

To apply AOP to existing systems and transform its crosscutting concerns to proper aspects, a maintainer needs to perform two tasks: one is the identification of crosscutting concerns in the system also known as aspect mining, the other is the actual refactoring of such concerns to a localized aspect. Because a maintainer is often not that familiar with all the details of the system, it is desirable to automate these tasks as much as possible. In this paper we focus on the second task and consider the case where one refactors the concerns to an aspect in an AspectJ-like language. One then must uncover suitable pointcuts that capture the points where the crosscutting concern originally occurred. To support this uncovering process we propose the use of inductive reasoning techniques.

In the remainder of this paper we first discuss in more detail the goal of modularizing crosscutting concerns in legacy code and the technique of inductive logic programming which we propose for the uncovering of pointcuts. We discuss a number of problems which are inherent in any attempt to use automatic techniques for uncovering pointcuts. We present an experiment in which ILP is used to uncover pointcuts for one specific kind of aspects which are implemented in legacy code as unique methods. The final sections of the paper discuss our experiment in more detail and the conclusions we draw from it.

*Research assistant of the Fund for Scientific Research - Flanders, Belgium (F.W.O.)

†Author funded by a doctoral scholarship of the "Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen)"

2 Goal

Pre-AOP legacy code is bound to contain badly modularized aspects which a maintainer of the system would like to factor out. In order to aid the maintainer in performing such a refactoring, tool support is required. The basic requirement on such a tool is that it allows one to identify code to factor out into an aspect and turn it into an advice and pointcut while preserving the behavior of the application. An additional requirement is of course that the resulting advices and pointcuts actually enhance the structure of the application by improving the -ilities [1]. Another consideration to take into account is that the maintainer of the system often lacks a profound knowledge of the implementation, thus such a tool would have to be largely automated. The most important and challenging part of this process, which is the focus of this paper, is the automatic generation of suitable pointcuts.

Essentially our goal is to undo the manual weaving process originally performed for certain concerns in legacy code and turn these into actual aspects. In pre-AOP days adding, for example, logging to the software was done by manually inserting calls to the logging facilities at the necessary places in the code. Nowadays, however this kind of behavior can be implemented by using aspect technology. While adding the concern, the original developer of the software had a certain intention. While one can ignore this intention and use a pointcut that simply enumerates the locations in the code where the refactored concern originally occurred, the resulting pointcut does not at all improve the -ilities of the code. A better solution is to try and uncover the intention and encode it in the pointcut. Due to the improved comprehensibility of this pointcut, it is easier to maintain and evolve the software. However, as the maintainer of the code has limited understanding of the code, this process of extracting a pointcut has to be automated.

3 Inductive Logic Programming

A first question is what basic technique can be used in the first place to uncover intensional[1] pointcuts. We do not want to simply use a pointcut that enumerates all these points, but rather one that better reflects the intent of the original developer by uncovering a certain pattern that all these points have in common. Stated this way this is in fact a *data mining* problem, and we have looked into this field and the related field of *machine learning* [6] for potential solutions.

Inductive logic programming (ILP) [7] was chosen as a particularly apt technique, as we are using our previously developed AspectJ-like crosscut language based on logic meta programming [3]. Crosscut expressions are formulated in this Prolog-like language as logic queries over all the joinpoints occurring in a program. The core of the language consists of a number of logic predicates which can be used to express conditions on the joinpoints, similarly to AspectJ's pointcut designators.

ILP is an inductive reasoning technique that is situated at the intersection of logic programming and machine learning. The goal of the technique is to derive a logic query for a desired set of solutions, based on a background theory that provides information about these solutions. The technique also requires a set of negative examples which should not be solutions of the query. ILP will generalize over the solutions using the information from the background theory while using the negative examples to prevent over-generalization. The way in which these generalizations are performed is dependent on the chosen ILP algorithm.

We have so far successfully applied ILP to the problem of uncovering intensional descriptions of software views [9]. Software views in this work are logic queries defining a set of software entities that belong to a particular view over the software. ILP was used in an experiment where a set of state changing methods on a `Buffer` class were given as the desired set of solutions. We were able to successfully derive rules stating that all these methods belonged to the set because they did assignments to instance variables or recursively called a method which changed the state of the buffer.

The ILP technique maps well to the task of uncovering pointcut definitions. The desired solutions

[1] Notice that there is a distinction between "intentional" and "intensional". The word "intentional" reflects the intention as meant by a developer, while "intensional" has a set-theoretic meaning which indicates a concise description of the elements of a set. We make this distinction as we can not assert that the intention of the developer is reflected in the automatically derived intension of the pointcut.

for the ILP algorithm consists of the joinpoints on which we want to weave. The background theory contains information on these joinpoints, which consists of the solutions of the basic crosscut predicates. We should also include information about the method in which the joinpoints are defined like naming conventions, messages which are sent, ... The negative examples are implicitly defined as all the other joinpoints in the system.

The basic principle of ILP is generalization by introducing variables in conditions over the background information. Note that variables and unification in logic programming are fairly equivalent with the notion of wildcards in AspectJ [3], and that this construct is largely responsible for allowing pointcuts to be more concise [2]. As an example take the situation where the desired joinpoints both occur in the same class, though not in the same method. This means the background theory contains the facts `within(jp1, ClassA, selectorA)` and `within(jp2, ClassA, selectorB)`. The ILP algorithm will generalize these two facts into a single condition stating `within(?jp, ClassA, ?selector)`, where `?jp` and `?selector` are variables, thus having uncovered a pattern in these joinpoints.

4 Aspect Mining

We have identified interesting code to factor out into an aspect using a comparatively simple technique. Some existing aspect mining techniques [4] have taken a clue from the observation that scattered code duplication is often a symptom of tangled concerns [5], and have hence used pattern matching techniques to find similar code fragments across a system. We have based ourselves on a somewhat more limited observation: when one implements the kind of aspects that are well handled by an AspectJ-like language without actually using such a language, one winds up implementing an "advice" as a uniquely named method invoked from several places in the system.

We applied the search for unique methods on a standard VisualWorks Smalltalk image, which contained about 3400 classes implementing a total of about 66000 methods. The results were sorted by the number of messages invoking each method and the list was then scanned manually for methods that could be factored out as aspects. The manual scan was necessary to filter out unique methods that are used to perform side-effects that are part of the basic functionality of the code and would not be interesting to factor out. The methods potentially implementing aspects were mostly identified on the basis of certain keywords in their name, and by looking at the messages that invoke them.

Class	Selector
`Parcel`	`#markAsDirty`
`ParagraphEditor`	`#resetTypein`
`UIPainterController`	`#broadcastPending-SelectionChange`
`CodeRegenerator`	`#pushPC`
`AbstractChangeList`	`#updateSelection:`
`PundleModel`	`#updateAfterDo:`

Table 1: Classes and selectors of identified aspects.

Table 1 shows the implementing classes and names of a few unique methods which we identified as potential aspects. All of these turned out to be examples of "update notification" aspects, apart from the `markAsDirty` method which is an example of a "cache invalidation" aspect.

To our surprise another well-known example of an "update notification" aspect in Smalltalk, the dependency mechanism involving the `changed:` method, did not turn up in our unique methods list. The `changed:` method is provided by the `Object` class and can be used by any object to notify its dependents of it having changed. Two other classes override this method however to add logging and screen updating respectively. This shows the limitation of the simple mining technique used. Also cases where for example two methods `logToServer:` exist because there are two variants of the logging aspect will have been missed by this technique.

The limitations of the aspect mining technique are however not that important, as our focus in this paper is on the uncovering of intensional pointcuts. By limiting ourselves to factoring out single statements invoking unique methods we can focus on the problem of automatically uncovering pointcuts to weave such statements back into the code. The problems discussed later on are independent of the mining technique used.

```
methodA
  self methodB.
  self log: variable.
  self methodC.

methodB
  variable := 'abc'.
  variable2 := 'def'.

methodC
  variable2 := 'ghi'.
```

Figure 1: Smalltalk source for the example joinpoint representation.

```
reception jp methodA
  message jp methodB
    reception jp methodB
      assignment jp variable
      assignment jp variable2
  access jp variable
  message jp log:
     reception jp log:
       ...
  message jp methodC
    reception jp methodC
      assignment jp variable2
```

Figure 2: Example representation of joinpoints.

```
(1) reception jp methodA
(2)   message jp methodB
(3)      reception jp methodB
(4)        assignment jp variable
(5)        assignment jp variable2
      *
(6)   message jp methodC
(7)     reception jp methodC
(8)       assignment jp variable2
```

Figure 3: Joinpoint graph after deleting the logging code.

5 General Issues

We have identified several issues which are inherent in any attempts to uncover an intensional description for a pointcut. In discussing these issues we will show a few example joinpoint graphs using the textual representation exemplified in figure 2. The figure shows the joinpoints occurring when method `methodA` shown in figure 1 is executed. The joinpoints are listed one after another as they occur in time, while indentation is used to show that a joinpoint occurs in the cflow of another. The ellipses shows where the joinpoints of whatever method invoked by the message `log:` fit in.

When deleting a code snippet from a method, as the first step in the refactoring to an advice, the joinpoints that result as the consequence of that snippet are removed from the joinpoint graph of every execution of the program. For example when we remove the logging statement from figure 1, the joinpoint graph of the execution of method `methodA` becomes the one shown in figure 3. The asterisk indicates the point at which an advice should reinsert the joinpoints of the removed snippet.

When refactoring all calls to a unique method there will be several such "asterisks". Our goal is to uncover a pointcut for all of them, but in the following sections we will focus on one such point that causes problems which render it difficult to uncover an intensional pointcut for the set of all of those points.

5.1 Joinpoint Choice Problem

A first problem in determining a pointcut to inject joinpoints back into the joinpoint graph at a certain position is that either the joinpoint right before or the one right after that position can be intercepted. That is, the refactored advice could be either a before- or an after-advice. For either choice of advice there are often even other joinpoints besides the one right before or after the position that could be intercepted.

Taking figure 3 as an example again, there are several advices with different pointcuts possible to inject logging at the point denoted by the asterisk. The advice could specify that logging should occur after the joinpoint labeled (2) or before the joinpoint labeled (6). In the after case, the advice could in fact also intercept either joinpoint (5) and in the before case also the joinpoint (7) or (8). While weaving the logging code on either of these three joinpoints does not result in the exact same joinpoint graph as the original in figure 2, the effect of the refactored code would in fact be the same.

```
self log: 'About to execute
           methodD:with: now.'.
self methodD: self computedValue
     with: self anotherComputedValue
```

Figure 4: Example source code for flattened expressions problem.

```
*
message jp computedValue
  ...
message jp anotherComputedValue
  ...
message jp methodD:with:
  ...
```

Figure 5: Example joinpoint graph for flattened expressions problem.

The refactoring should not only preserve the behavior of the application, but the generated pointcut also should improve the -ilities of the software. This has an impact on which of the possible joinpoints the generated pointcut should be based on. As in the example, when the intent was for logging to occur in certain methods such as `methodA` after the message `methodB` was sent, the pointcut should definitely be based on joinpoint (2), while on the other hand if the intent was for logging to occur before sends of the message `methodC`, the joinpoint (6) should be used. Choosing the wrong joinpoint would result in an incorrect pointcut that aggravates the maintainer's problems.

5.2 Flattened Expressions Problem

A second problem in determining pointcuts is caused by the fact that the subexpressions of an expression result in a flattened list of joinpoints preceding the joinpoint of the expression itself. Take the code of figure 4, the joinpoint graphs resulting from executing that code will have the structure shown in figure 5. It is clear from reading the source that the intention of the original developer was for logging to occur because the `methodD:with:` method is called. When the developer had used aspect-oriented programming, she would have specified a pointcut intercepting this

```
   ...
self cacheTimedOut
       ifTrue:[self invalidateCache]
   ...
```

Figure 6: Example of the Dynamic Conditions & Properties Problem.

```
after ?jp matching {
    ...,
    inObject(?jp,?obj),
    [?obj cacheTimedOut] }
```

Figure 7: A possible pointcut for the example above.

call to `methodD:with:`. However this is not exactly reflected in the manually woven implementation as can be seen in the joinpoint graph: the logging actually precedes the `computedValue` message joinpoint. This is not an error of the original developer but a compromise between having the logging happen at the exact correct moment and the clarity of the code: while the developer could have used temporary variables to store the arguments of `methodD:with:`, send the logging and then call the method using the temporaries as arguments, this would have cluttered the code and is therefore not common coding practice. This unfortunately complicates the automated uncovering of a pointcut and advice, as the tool can not always determine whether weaving on the `methodD:with:` message joinpoint instead of the `computedValue` message joinpoint would not alter the desired behavior of the application. The only option for the tool is then to use a pointcut weaving on the `computedValue` message joinpoint. There may be several other occurrences of logging before a `methodD:with:` message where the tool has to actually weave on the joinpoint defining the value of the `with:` argument. This unfortunately does not reflect the original intent of the developer and will probably result in a less-than ideal pointcut.

5.3 Dynamic Conditions Problem

While less a fundamental problem than the previous two issues, a third issue that needs to be taken into account is that there are sometimes dy-

namic conditions attached to the manually woven concerns. Figure 6 shows an example where the method `invalidateCache` is only invoked when the cache has timed out. It would not clarify the code to only refactor the `invalidateCache` message send while leaving the `cacheTimedOut` condition in place. The test should in fact be part of the pointcut itself.

Figure 7 depicts the structure of a possible pointcut. Note that the condition `cacheTimedOut` is part of the pointcut which will only weave if the receiving object's cache is timed out. This condition is specified using a boolean Smalltalk expression enclosed in square brackets, this is equivalent to AspectJ's `if` mechanism.

Unlike the previous two issues, this issue is probably straightforward to solve. When identifying snippets of code as implementing an aspect, one needs to check whether this code occurs in a condition and is the only consequent of that condition. Putting the condition in the pointcut itself requires transforming variable references such as the one to `self` in the example to variables in the pointcut such as the `?o` variable which captures the value of `self` in the original context of the joinpoint using the `inObject` predicate.

6 Applying ILP

Before discussing an experiment we performed, we will first give a brief overview of how the input for the ILP algorithm is determined and what specific ILP algorithm we have used in our experiments. To determine the joinpoints on which to weave to refactor calls to unique methods, we first construct a skeleton joinpoint graph similar to the one shown in figure 2 of the program. This joinpoint graph covers the possible joinpoint graphs that can actually occur when executing the code. The properties of the joinpoints in this graph are limited to information that can be statically determined from the program's source. From the graph we determine the joinpoints following and preceding the call that will be refactored, these joinpoints can be used to respectively weave before or after. For all these joinpoints, the facts of the predicates in the crosscut language are determined and used as the background information for the ILP algorithm. The joinpoints themselves form the desired solution set of the algorithm.

The specific ILP algorithm we used is based on relative least general generalization [8]. The algorithm creates queries by anti-unifying each input fact with all the other input facts. Anti-unification is simply the process of introducing variables into facts where they differ. The algorithm boils down to recognizing similarities in the background knowledge and using this information to create queries that describe the desired solutions.

7 Example in Smalltalk

In this section we take a closer look at one of the potential aspectual methods we identified in Smalltalk. This method is representative for some of the difficulties we encountered in applying ILP to uncover intensional pointcuts.

The method `broadcastPendingSelectionChange` is implemented on the class `UIPainterController`, a class in the VisualWorks UI Framework. There are 18 calls to this method from within several other classes of the framework, some of these calls are shown in figure 8 in the context in which they occur.

While our goal is to automatically uncover pointcuts describing these calls with as little input from the maintainer as possible, for this experiment we did first investigate what the intent of the method is. From the method's name we can guess it implements a notification aspect which notifies certain other objects that the current selection is about to change.

We first applied the ILP algorithm to the set of direct before and also the set of direct after joinpoints of the `broadcastPendingSelectionChange` calls. The ILP algorithm failed to uncover a logic query which covered all the desired joinpoints while not covering any negative examples. This means there is no better solution than to simply enumerate the joinpoints. The likely cause is that the direct before and after joinpoints are in fact not related to the intent of calling `broadcastPendingSelectionChange`. For example many of the direct after joinpoints are calls of `spec` occurring in the code in calls of `replaceElement:basedOnSpec:` as shown in figure 8. There are in fact many calls of `spec` in the Smalltalk image before which there is no call to

```
controller broadcastPendingSelectionChange.
controller select:  (controller replaceElement:  sels first basedOnSpec:self spec)
```

```
controller broadcastPendingSelectionChange.
controller removeSelection:  selection
```

```
controller broadcastPendingSelectionChange.
controller appendSelections:  selections
```

```
controller broadcastPendingSelectionChange.
currentState := self extractFullSpec.
```

```
doEmbed
 self broadcastPendingSelectionChange.
 (selectedComponents := self selectedComponents) size > 0 ...
 ...
 self removeSelections.
```

Figure 8: Examples of snippets in the code where `broadcastPendingSelectionChange` is used.

```
 if
send(?joinpoint,?message),
within(?joinpoint,[SelectModeTracker],?selector),
inProtocol([UIPainterController],?message,[#'selection manipulation'])
```

Figure 9: The induced pointcut

`broadcastPendingSelectionChange`. Our earlier investigation suggests that the call actually occurs because of the call to `select:` that follows it, a method that changes the selection. This again illustrates the flattened expressions problem.

A more severe variant of the flattened expression problem also occurs in practice. The last example in figure 8 shows a method `doEmbed` where `broadcastPendingSelectionChange` occurs at the beginning of the method, while an actual change of the selection doesn't happen until a few lines further down. We do not know whether this call *has* to occur at this location or whether it was implemented like this for aesthetical reasons.

In a second experiment, we ignored the flattened expressions problem and manually selected the after joinpoints which we presumed are related to the intent of the calls to `broadcastPendingSelectionChange`. Figure 9 shows the intension which is derived by the ILP algorithm. This logic query states that `broadcastPendingSelectionChange` should occur before every send joinpoint to a method implemented in the protocol `selection manipulation`. Although this query describes the intent of the pointcut fairly well, it also covers a small number of joinpoints which should not be woven on. This is due to our limited implementation of the ILP algorithm which does not add these joinpoints as exceptions or negated clauses to the induced query.

8 Discussion

Based on our experiments we can conclude that automatically uncovering pointcuts from legacy code using machine learning techniques is a promising approach, although the three problems we have discussed above need to be solved. A potential solution for the flattened expression problem is to extend dynamic joinpoint models with a new construct: the "statement joinpoint". A statement joinpoint would consider the execution of a statement as a single joinpoint, with all the joinpoints of the subexpressions of the statement occurring in its cflow. This bridges the mismatch between the dynamic joinpoint models used in AOP and the static weaving model used when manually weaving. This is exactly what we did in our second experiment where we ignored the flattened expressions and used the top-level message joinpoints of statements where `select:` etc. where sent. Using a

pointcut language supporting statement joinpoints would allow us to simply weave on executions of statements where the messages `select:` etc. are sent.

As for the dynamic conditions problem, a solution was already discussed when discussing the problem. While this problem is one to take into account, its solution seems straightforward by transforming the conditions to be part of the pointcut.

The joinpoint choice problem is more difficult to tackle. A potential solution we have in mind would be to apply ILP to all possible combinations of joinpoints and to compare the resulting pointcuts. We expect that when any of these combinations are related to the intent of why the aspect needs to be woven there, this will result in a more intensional pointcut than for the other combinations. This of course necessitates a way of comparing pointcuts based on their relative intensionality. Of course considering all possible combinations leads to a combinatoric explosion, but many combinations can be disregarded: it is uncommon to have an advice that needs to occur before some points and after others, to occur after the reception of some messages and after the last statement of a method, ...

9 Future Work

One of the obstacles we encountered in our experiment was due to the limited implementation of our ILP system and the particular algorithm used. The algorithm used never adds negative conditions to queries to ensure no negative examples are covered. In cases where the induced pointcut is too general and covers just a few negative examples, it might be better to explicitly add the negative examples as exceptions, rather than resort to an enumeration of the desired joinpoint solutions. Furthermore, there exist numerous different ILP algorithms. In our experiment we used one of these algorithms: relative least general generalization. Using other algorithms, or other machine learning techniques altogether, can lead to the induction of better pointcuts and may also prove to be more efficient.

In this paper we limited ourselves to aspects which are implemented as a unique method. As mentioned before, this approach is too restrictive to cover all possible aspects. Part of our future work consists of generalizing our approach and conducting experiments with other kinds of aspects.

It might also be interesting to try our approach on Java programs. Due to the static typing in Java, more information is available to the ILP algorithm which may lead to better pointcuts. On the other hand, we could also obtain this information in Smalltalk using a type inferencer. However then we would have to take into account that the inferred types are not necessarily correct.

10 Conclusions

We have demonstrated the refactoring of aspects implemented as unique methods using one particular technique to uncover intensional pointcuts: inductive logic programming. We first discussed a number of elementary problems that face any automatic uncovering technique: the joinpoint choice problem, the flattened expressions problem and the dynamic conditions problem. These problems prevented us from uncovering a suitable pointcut in the first of our experiments. In a second experiment we manually circumvented the problems and were able to uncover a pointcut that both described the correct joinpoints, and exhibited the desired -ilities because of its intensional description. In the discussion we proposed for future work automatic ways of circumventing the problems. Although several problems still need addressing, the automated uncovering of intensional pointcuts to support the aspectualization of legacy code seems to be a possibility.

Acknowledgments

We would like to thank Johan Brichau for proofreading this paper and his insightful comments.

References

[1] R. E. Filman. Injecting ilities. In *Proceedings of the Aspect-Oriented Programming workshop at ICSE'98*, 1998.

[2] R. E. Filman. Aspect-oriented programming is quantification and obliviousness. In P. Tarr, L. Bergmans, M. Griss, and H. Ossher, editors,

Proceedings of the Workshop on Advanced Separation of Concerns at OOPSLA 2000, 2000.

[3] 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.

[4] J. Hannemann. Overcoming the prevalent decomposition of legacy code. In *Workshop on Advanced Separation of Concerns at the International Conference on Software Engineering 2001*, 2001.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European conference on Object-Oriented Programming*. Springer-Verlag, jun 1997.

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

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

[8] G. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

[9] T. Tourwé, J. Brichau, A. Kellens, and K. Gybels. Induced intentional software views. Submitted to European Smalltalk Users Group conference 2003.