Inducing Evolution-Robust Pointcuts

Mathieu Braem, Kris Gybels, Andy Kellens*, Wim Vanderperren
System and Software Engineering Lab
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussels, Belgium
{mbraem,kgybels,akellens,wvdperre}@vub.ac.be

ABSTRACT

One of the problems in Aspect-Oriented Software Development is specifying pointcuts that are robust with respect to evolution of the base program. We propose to use Inductive Logic Programming, and more specifically the FOIL algorithm, to automatically discover intensional pattern-based pointcuts. In this paper we demonstrate this approach using several experiments in Java, where we successfully induce a pointcut from a given set of joinpoints. Furthermore, we present the tool chain and IDE that supports our approach.

1. INTRODUCTION

Aspect-Oriented Software Development (AOSD) aims to provide a better separation of concerns than possible using traditional programming paradigms [17]. To this end, AOSD introduces an additional module construct, called an *aspect*. Traditional aspects consist of two main parts: a *pointcut* and an *advice*. Points in the program's execution where an aspect can be applied are called *joinpoints*. Pointcuts are expressions in a pointcut language which describe a set of joinpoints where the aspect should be applied. The advice is the concrete behavior that is to be executed at a certain joinpoint, typically before, after or around the original behavior at the joinpoint.

Since existing software systems can benefit from the advantages of AOSD as well, a number of techniques have been proposed to identify crosscutting concerns in existing source code (aspect mining) [4, 5, 6] and transform these concerns into aspects (aspect refactoring) [21, 20]. When refactoring a concern to an aspect, a pointcut must be written for this aspect. Pointcut languages like for instance the CARMA pointcut language allow specifying pattern-based pointcuts, so that the pointcut does not easily break when the base code is changed [9, 18]. While existing aspect refactoring techniques also automatically generate a pointcut, they typically only provide an enumerative pointcut, which is fragile with respect to evolution of the base program. Turning this pointcut into a pattern-based pointcut is left to be done manually by the developer.

In this paper we propose to exploit Inductive Logic Programming techniques to automatically induce a pattern-based pointcut from a given set of joinpoints. The next section details the problem of uncovering pattern-based pointcuts and introduces the running example used throughout this paper. Section 3 introduces Inductive Logic Programming and the concrete algorithm used and in section 4 we apply ILP for automatically generating pattern-based pointcuts and report on several successful experiments in Java. Afterwards, we present the tools created to support our approach, compare with related work and state our conclusions.

2. MOTIVATION AND BACKGROUND

The main problem in maintaining aspect-oriented code is the so-called *fragile pointcut problem* [19]. Pointcuts are deemed fragile when seemingly innocent changes to a program result in the pointcut no longer capturing the intended joinpoints. Taking the code of figure 1 for example, a pointcut for capturing message invocations on Point objects that change the state of the object could simply say "capture `setX` and `setY` messages". Changing the name of `setX` to `changeX` or adding a method `setZ` would break this pointcut. The pointcut is obviously fragile because it is simply an enumeration of methods.

Using an advanced pointcut language that gives access to the full static joinpoint model of a program, it is possible to exploit a more robust pattern [9]. Figure 2 illustrates a pointcut that expresses that all the state changing methods contain an assignment to an instance variable of an object.

The area of aspect refactoring and aspect mining is a particularly interesting research area within AOSD that is currently being explored. In performing aspect mining and refactoring, the problem crops up of finding a pointcut for the newly created aspect. Also, as with object-oriented refactoring, research is being performed on how to automate these refactorings using tool support. In such tools, it would be interesting to be able to automate the step of generating a pattern-based pointcut as well. Currently, most proposals for automating aspect refactoring simply generate an enumerative pointcut, which is of course too fragile. In related

*Ph.D. scholarship funded by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen).

```
1  public class Point {
2
3      private int x,y;
4
5      public void setX(int a) {
6          this.x=a;
7      }
8      public void setY(int a) {
9          this.y=a;
10     }
11     public int getX() {
12         return x;
13     }
14     public int getY() {
15         return y;
16     }
17 }
```

Figure 1: A simple Point class

```
1  stateChanges(Jpvar):
2      execution(Jpvar,MethodName),
3      inMethod(AssignmentJP,MethodName),
4      isAssignment(AssignmentJP,AssignmentTarget),
5      instanceVariable(AssignmentTarget,ClassName)
```

Figure 2: A pointcut for the observer

work, section 6, we discuss a number of these approaches.

In this paper, we therefore propose the use of Inductive Logic Programming for automatically generating a pattern-based pointcut. We restrict ourselves to a pointcut language with a static joinpoint model. We in particular use a logic pointcut language similar to CARMA, but restricted to a static joinpoint model, which means that run-time values and cflow constructs are not supported.

3. INDUCTIVE LOGIC PROGRAMMING

The technique of logic induction returns a logic query that, by using conditions drawn from background information on a set of examples, satisfies all positive examples while not including any negative examples. In this paper we use the FOIL ILP algorithm [23]. Informally, the way ILP works and how it can be applied to generate a pattern-based pointcut is as follows:

positive examples: ILP takes as input a number of positive examples, in our setting of deriving pattern-based pointcuts these are joinpoints that the pointcut should capture.

negative examples: ILP also takes as input a number of negative examples, the rules that are derived during the iterative induction should never cover negative examples. Negative examples effectively force the algorithm to use other information of the background in the induced rules.

background information: Another input to ILP is background information on the examples. In our setting, these would be the result of predicates in the pointcut language that are true for the joinpoints, or in other words, the data associated with the joinpoints. Such as the name of the message of the joinpoint, the type of the joinpoint (message, assignment, ...), in which method or class the joinpoint occurs, ...

induction: ILP follows an iterative process for constructing a logic rule based on the positive examples. Starting from an empty rule, in each step of the process, the rule is extended with a condition drawn from the background information which decreases the number of negative examples covered by the rule. This is repeated until the rule describes all positive but no negative examples. The added conditions are generalizations of facts in the background information, by adding logic variables (a more powerful version of wildcards). For example, if the background information contains the fact `instanceVariable('Point.x', 'int')` one possible condition used could be `instanceVariable(X, 'int')`.

4. ILP FOR POINTCUT ABSTRACTION

In this section, we perform a number of experiments in order to demonstrate how we use the FOIL algorithm to induce pattern-based pointcuts. The joinpoints required as positive examples for the ILP algorithm can be selected automatically using for example an aspect mining technique, though in these experiments we selected them manually. All other joinpoints are defined as negative examples for the ILP algorithm. As background information, we construct a logic database consisting of the information that is normally available in the pointcut language on these joinpoints and structural information about the program, such as the relationship between classes etc. Because the pointcut language uses a purely static joinpoint model, these solutions can be determined using only the program's source or compiled representation, i.e. compiled Java classes. Examples of these facts are given in figure 3.

The algorithm will induce a pointcut that captures exactly the joinpoints currently in the program that should be captured (the positive examples), and none of the others (the negative examples). This is guaranteed by the algorithm. It is reasonable to expect, though not guaranteed, that the induced pointcut also is a non-fragile or robust pointcut because the induction process generalizes the conditions it adds to rules. In general we will not have a specific pointcut in mind that the algorithm should derive (otherwise the application of ILP would be rather pointless), though in these experiments we can use the robust pointcut from figure 2 as a benchmark for comparison. We do not discuss the performance of our tools, but an analysis is included in an extended version of this paper [3].

4.1 Basic Point class

As an example of our approach, take the simple Point class from figure 1. In a first step we derive the static joinpoints from this code, and derive the information on all of these that is given by the predicates of the pointcut language. This forms the background information for the logic induction algorithm, part of this generated background information is shown in figure 3.

The methods that are state changing on this simple Point class are the methods `setX` and `setY` only. We identify these two joinpoints as positive examples of our desired

```
returnStatement(jp1).
returnStatement(jp6).
returnStatement(jp11).
returnStatement(jp14).
returnStatement(jp17).
inMethod(jp1,'Point.setX(I)I').
inMethod(jp2,'Point.setX(I)I').
inMethod(jp3,'Point.setX(I)I').
inMethod(jp4,'Point.setX(I)I').
inMethod(jp6,'Point.setY(I)I').
inMethod(jp7,'Point.setY(I)I').
inMethod(jp8,'Point.setY(I)I').
inMethod(jp9,'Point.setY(I)I').
inMethod(jp11,'Point.getX()I').
inMethod(jp12,'Point.getX()I').
inMethod(jp14,'Point.getY()I').
inMethod(jp15,'Point.getY()I').
inMethod(jp17,'Point.Point()V').
isRead(jp3,'l0').
isRead(jp4,'l1').
isRead(jp8,'l2').
isRead(jp9,'l3').
isRead(jp12,'Point.x').
isRead(jp15,'Point.y').
methodInClass('Point.setX(I)I','Point').
methodInClass('Point.setY(I)I','Point').
methodInClass('Point.getX()I','Point').
methodInClass('Point.getY()I','Point').
methodInClass('Point.Point()V','Point').
classExtends('Point','java.lang.Object').
methodReturns('Point.setX(I)I','int').
methodReturns('Point.setY(I)I','int').
methodReturns('Point.getX()I','int').
methodReturns('Point.getY()I','int').
isAssignment(jp2,'Point.x').
isAssignment(jp7,'Point.y').
instanceVariable('Point.x','Point,int').
instanceVariable('Point.y','Point,int').
classInPackage('java.lang.Object','java.lang').
execution(jp0,'Point.setX(I)I').
execution(jp5,'Point.setY(I)I').
execution(jp10,'Point.getX()I').
execution(jp13,'Point.getY()I').
execution(jp16,'Point.Point()V').
```

Figure 3: Part of the background information for the Point class of figure 1.

```
1 stateChanges(A):
2     execution(A,B),
3     inMethod(C,B),
4     isAssignment(C,D).
```

Figure 4: Induced stateChanges pointcut.

`stateChanges` pointcut, which are the joinpoints jp0 and jp5 respectively. The pointcut should not cover the other joinpoints: the joinpoints jp10 and jp13, for instance, denote the execution of the `getX` and `getY` method. Clearly, these methods are not state changing. So these and all other joinpoints besides jp0 and jp5 are marked as negative examples. We give the FOIL algorithm the positive examples `stateChanges(jp0)` and `stateChanges(jp5)`. The resulting rule is shown in figure 4. The pointcut selects all executions of methods that contain an assignment.

The resulting pointcut is clearly not very robust. An evolution that easily breaks the pointcut would be to have a `getX` method that does an assignment to a local variable which does not mean that that method changes the state of an object, yet its execution would be captured by the pointcut. This result is however not very surprising: the Point class is small and does not include non-state changing methods that do assignments to local variables which would have served as a negative example for the FOIL algorithm. As the induced pointcut covers all positive examples and no negative ones, the induction stops and no further predicates from the background information are used to limit the rule to only the positive examples. The ILP algorithm works better on larger programs, so that more negative examples are available to avoid oversimplified pattern-based pointcuts.

In order to have a more realistic example, we apply our experiment to the Point class bundled with Java. We do not include a full listing of the generated background, but instead we give some statistics about the generated facts. Table 1 compares the number of facts found in the AWT Point class to the number of facts from the basic Point example. We notice a rapid increase in the number of facts with larger input. However, since the generated information is based on a static joinpoint model, this number only grows lineary, in function of the size of the classes and the size of the methods.

Table 1: Generated facts statistics

	# Classes	# Facts	# Joinpoints
Toy example	1	71	10
AWT Point class	1	364	70
Full AWT library	362	276863	65060

```
1 stateChanges(A):
2     execution(A,B),
3     inMethod(C,B),
4     isAssignment(C,D),
5     instanceVariable(D,E).
```

Figure 5: Resulting pointcut when applying our approach to the AWT Point class.

We identify four execution joinpoints in the AWT Point class where a state changing method is invoked and input them as positive examples to the algorithm. The remaining 66 joinpoints are defined as negative examples. The resulting pointcut is shown in figure 5. In this case, the algorithm generates a pointcut that is sufficiently robust for evolution: it is in fact the same pointcut we defined in figure 2.

4.2 Extended experiments

```
1 stateChanges(A):
2     execution(A,B),
3     inMethod(C,B),
4     isAssignment(C,D),
5     instanceVariable(D,E),
6     not(isTransient(D)).
```

Figure 6: Resulting pointcut for non-transient field assignments in Java AWT.

In order to provide a limited evaluation of our approach, we conduct two more involved experiments using the state-changes example on the Java AWT framework.

4.2.1 *Large fact database:*

We apply our approach to the complete Java AWT library in order to evaluate whether it still returns a useful result when the number of facts is very large. This library contains approximately 362 classes and generates more than 250000 facts. The result for the algorithm is the same as for the Java AWT Point class alone: the same pointcut as we defined in figure 2 is induced.

4.2.2 *Negation:*

One of the distinguishing features of the FOIL algorithm in comparison to other ILP algorithms is its ability to induce rules containing negations. As a variation of the state changing methods example, we need a pointcut for the executions of methods that change the observable representation of an object. This means the method does assignments to instance variables that are not declared transient using the modifier `transient` in Java: conceptually, these fields are not part of the object's persistent state and are not retained in the object's serialization. This is used for example when a class defines a cache in order to optimize some parts of its operations. As such, observers do not need to be notified when transient fields are altered. When applying this experiment to the Java AWT library, our algorithm induces the rule shown in figure 6, which in comparison to the pointcuts induced above adds exactly the properties in the background to distinguish these joinpoints from the negative examples that we would expect it to add, i.e. the fact that the instance variables being assigned to are not declared transient.

5. TOOL SUPPORT

5.1 Tool Chain

Our approach is supported by a fully automatic tool chain, consisting of the following tools (see figure 7):

- *FactGen:* This tool translates a range of Java class files and/or jar files to a set of facts representing these classes. The tool uses the javassist library [7] to process the binary class files. The javassist library provides a high-level reflective API that allows to inspect the full Java byte code, including method bodies. The output of the FactGen tool is the fact representation in XML format.
- *JFacts:* This tool allows to translate logic predicates from one syntax into another. Currently, the tool supports the FactGen's XML syntax, QFoil's syntax, CARMA's syntax and the Prolog syntax.

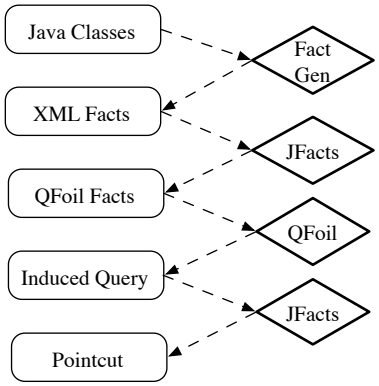

Figure 7: The tool chain for inducing pointcuts from Java classes.

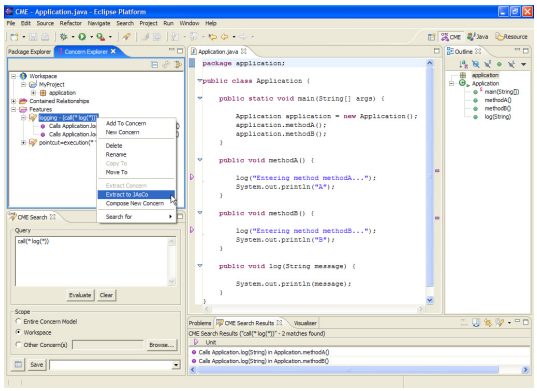

Figure 8: Screenshot of the Eclipse CME plugin that allows automatic concern extraction to JAsCo.

- *QFoil:* This tool is the implementation of the FOIL ILP algorithm by Ross Quinlan [24]. It takes a set of facts and a set of positive examples as input (negative examples are implicitly assumed) and tries to induce a logic rule that covers all of the positive examples and rejects all of the negative examples.

5.2 IDE Integration

Although our current tool chain works fully automatic, it is a stand-alone command-line tool that is not integrated in an IDE. In previous work, we have developed an AO refactoring extension to the Eclipse Concern Manipulation Environment [26]. Figure 8 shows a screenshot of this tool. This visual tool allows to navigate through applications via a powerful and extensible query language. As such, cross-cutting concerns can be identified and isolated in a concern model. Afterwards, the concern can be automatically refactored to an aspect in the JAsCo language [27]. However, the generated pointcut is simply an enumeration of joinpoints. Because JAsCo also supports the pointcut language presented in this paper, integrating our tool chain in this visual IDE is trivial. As such, a fully automatic refactoring tool can be realized that generates evolution-robust pointcuts instead of plain enumerations.

6. RELATED WORK

To our knowledge, there exist no other approaches which try to automatically generate pattern-based pointcuts. In previous work [10] we already report on a first attempt for using Inductive Logic Programming in order to derive pattern-based pointcuts. In this work we employ Relative Least General Generalisation [22], an alternative ILP algorithm, instead of the FOIL algorithm. Using RLGG, we are able to derive correct pointcuts for some specific crosscutting concerns in a Smalltalk image. However, due to the limitations of both our implementation as well as the applied ILP algorithm (for instance, the algorithm does not support negated literals), our RLGG-based technique often results in pointcuts that suffer from some fragility: the resulting pointcuts for example frequently contain redundant literals referring to the names of specific methods or classes, which of course easily breaks the pointcut when these names are changed. Furthermore, our earlier work suffers from serious scalability issues.

As mentioned earlier, the major area of application of our technique lies in the automated refactoring of crosscutting concerns in pre-AOP code into aspects. Quite a number of techniques exist [11, 21, 20, 13] which propose refactorings in order to turn object-oriented applications into aspect-oriented ones. However, these techniques do not consider the generation of pattern-based pointcuts. Instead they propose to automatically generate an enumeration-based pointcut which, optionally, can be manually turned into a pattern-based pointcut by the developer. As is pointed out by Binkley et al. [2], our technique is complementary with these approaches as it can be used to both improve the level of automation of the refactoring, as well as the evolvability of the refactored aspects.

In the context of aspect mining, which is closely related to object-to-aspect refactorings, a wealth of approaches are available that allow for the identification of crosscutting concerns in an existing code base. The result of such a technique is typically an enumeration of joinpoints where the concern is located. Ceccato et al. [6] provide a comparison of three different aspect mining techniques: identifier analysis, fan-in analysis and analysis of execution traces. Breu and Krinke propose an approach based on analyzing event traces for concern identification [4]. Bruntink et al. [5] make use of clone detection techniques in order to isolate idiomatically implemented crosscutting concerns. Furthermore, several tools exist that support aspect mining activities by allowing developers to manually explore crosscutting concerns in source code, such as the aspect mining tool [12], FEAT [25], JQuery [15] and the Concern Manipulation Environment [14]. These approaches are complementary with our approach in that the joinpoints they identify can serve as positive examples for our ILP algorithm.

7. CONCLUSIONS AND FUTURE WORK

In this paper we present our approach using Inductive Logic Programming for generating a concise and robust pointcut from a given enumeration of joinpoints. We report on a number of successful experiments that apply our approach to a realistic and medium-scale case study.

In future work we will consider tackling full CARMA which requires taking into account in the background information that joinpoints and joinpoint shadows are not equated as in the more restricted pointcut language used here. Our approach can easily be applied to for example AspectJ [16] as well by translating the induced pointcuts to AspectJ pointcuts. However, the FOIL algorithm must then be restricted to not generate pointcuts using features that can not be translated to AspectJ: variables can only be used once in a pointcut (except when using the "if" restrictor in AspectJ), recursive named pointcuts are not possible, and only some uses of the structural predicates can be translated. Other points left for future work are:

- *Other Algorithms:* There exist several algorithms for Inductive Logic Programming. In previous work, we conduct several small-scale experiments with the Relative Least General Generalization (RLGG) [22] algorithm in an aspect mining context [10]. Having several algorithms might improve the quality of the selected results to the end-user. For example, solutions that are induced by more than one algorithm might be better.

- *Multiple Results:* Our current tools only generate one pointcut for a given set of joinpoints. In some cases, most notably when there is little background information (i.e. a small number of little classes), several alternative pointcuts are possible. Therefore, it would be useful to allow presenting multiple pointcut results.

- *Run-Time Information:* Our current approach only analyzes the static program information to induce pointcuts. Pointcuts that require run-time program information, such as stateful aspects [8], cannot be induced. For this, facts representing the run-time behavior of the program are necessary.

- *Refactor existing pointcuts:* The FOIL ILP algorithm could refactor pointcuts given by an aspect-mining technique. This can be easily done by simply taking all the joinpoints covered by that pointcut and using them as the positive examples.

8. REFERENCES

[1] Mehmet Akşit, editor. *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003).* ACM Press, March 2003.

[2] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. Automated refactoring of object oriented code into aspects. In *21st IEEE International Conference on Software Maintenance (ICSM)*, 2005.

[3] Mathieu Braem, Kris Gybels, Andy Kellens, and Wim Vanderperren. Automated pattern-based pointcut generation. In *Proceedings of Software Composition*, LNCS. Springer-Verlag, 2006. (to appear).

[4] Silvia Breu and Jens Krinke. Aspect mining using event traces. In *19th International Conference on Automated Software Engineering*, pages 310–315, Los Alamitos, California, September 2004. IEEE Computer Society.

[5] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. An evaluation of clone detection

techniques for identifying crosscutting concerns. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*. IEEE Computer Society Press, 2004.

[6] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonello, and T. Tourwé. A qualitative comparison of three aspect mining techniques. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC 2005)*, pages 13–22. IEEE Computer Society Press, 2005.

[7] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In *GPCE '03: Proceedings of the second international conference on Generative programming and component engineering*, pages 364–376, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[8] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In Karl Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, pages 141–150. ACM Press, March 2004.

[9] Kris Gybels and Johan Brichau. Arranging language features for pattern-based crosscuts. In Akşit [1], pages 60–69.

[10] Kris Gybels and Andy Kellens. An experiment in using inductive logic programming to uncover pointcuts. In *First European Interactive Workshop on Aspects in Software*, September 2004.

[11] Stefan Hanenberg, Christian Oberschulte, and Rainer Unland. Refactoring of aspect-oriented software. In *4th Annual International Conference on Object-Oriented and Internet-based Technologies,Concepts, and Applications for a Networked World*, 2003.

[12] J. Hannemann. The Aspect Mining Tool web site. http://www.cs.ubc.ca/labs/spl/ projects/amt.html.

[13] Jan Hannemann, Gail Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. In Peri Tarr, editor, *Proc. 4rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2005)*, pages 135–146. ACM Press, March 2005.

[14] William Harrison, Harold Ossher, Stanley M. Sutton Jr., and Peri Tarr. Concern modeling in the concern manipulation environment. IBM Research Report RC23344, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, September 2004.

[15] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In Akşit [1], pages 178–187.

[16] 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.

[17] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[18] Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *European Conference on Object-Oriented Programming, ECOOP 2005*, 2005.

[19] Christian Koppen and Maximilian Störzer. PCDiff: Attacking the fragile pointcut problem. In Kris Gybels, Stefan Hanenberg, Stephan Herrmann, and Jan Wloka, editors, *European Interactive Workshop on Aspects in Software (EIWAS)*, September 2004.

[20] Ramnivas Laddad. Aspect-oriented refactoring, dec 2003.

[21] Miguel Pessoa Monteiro. Catalogue of refactorings for aspectj. Technical Report UM-DI-GECSD-200401, Universidade Do Minho, 2004.

[22] S. Muggleton and C. Feng. Efficient induction in logic programs. In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, 1992.

[23] J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, August 1990.

[24] Ross Quinlan. Qfoil: the reference foil implementation. Home page at `http://www.rulequest.com/Personal/`, 2005.

[25] Martin P. Robillard and Gail C. Murphy. Automatically inferring concern code from program investigation activities. In *Proceedings of Automated Software Engineering (ASE) 2003*, pages 225–235. IEEE Computer Society, 2003.

[26] Davy Suvée, Bruno De Fraine, Wim Vanderperren, Niels Joncheere, Len Feremans, and Karel Bernolet. JAsCoCME: Supporting JAsCo in the Concern Manipulation Environment. In *CME BOF at the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, Chicago, IL, USA, March 2005.

[27] Davy Suvée and Wim Vanderperren. JAsCo: An aspect-oriented approach tailored for component based software development. In Akşit [1], pages 21–29.