Automated Pattern-Based Pointcut Generation

Mathieu Braem, Kris Gybels, Andy Kellens* Wim Vanderperren

Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium
{mbraem,kgybels,akellens,wvdperre}@vub.ac.be

Abstract. One of the main problems in Aspect-Oriented Software Development is the so-called fragile pointcut problem. Uncovering and specifying a good robust pointcut is not an easy task. In this paper we propose to use Inductive Logic Programming, and more specifically the FOIL algorithm, to automatically identify intentional pattern-based pointcuts. We present the toolchain we implemented to induce a pointcut given a set of identified joinpoints. Using several realistic medium-scale experiments, we show that our approach is able to automatically induce robust pointcuts for a set of joinpoints.

1 Introduction

Separation of concerns [27] is a crucial property for realizing comprehensible and maintainable software. Current software engineering paradigms do however not always succeed in cleanly modularizing all concerns. Consequently, these concerns are spread and repeated over several modules in the system. Due to this code duplication, it becomes very hard to alter such concerns within the system. These concerns are called *crosscutting* because the concern virtually crosscuts the decomposition of the system. Typical examples of crosscutting concerns are debugging concerns such as logging [19] and contract verification [31], security concerns [8] such as confidentiality and access control, and business rules [26, 9] that describe business-specific logic.

Aspect-Oriented Software Development aims to provide a solution for these crosscutting concerns [19]. To this end, AOSD introduces an additional module construct, named an *aspect*. Traditional aspects consist of two main parts: a *pointcut* definition and an *advice*. Points in the program's execution where an aspect can be applied are called *joinpoints*. The declarative pointcut language allows to concisely 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 pointcut, typically before, after or around the original behavior identified by the joinpoints.

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 source code (aspect mining) and transform these concerns into aspects (aspect refactoring). Although these aspect refactorings are automated to a certain degree, the resulting pointcuts only provide an enumeration of the joinpoints

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

which need to be captured by the aspect. As a result, the slightest change in the base code may easily break the pointcut. Although solutions to this problem exist, like for instance the CARMA pointcut language [11] that allows to specify pointcuts that describe collections of joinpoints more intentionally, the job of rendering the refactored pointcut more intentional has to be done manually by the developer.

In this paper we propose to exploit Inductive Logic Programming techniques to automatically deduce an intentional pointcut from a given set of joinpoints. The next section details the problem of uncovering intentional pointcuts and introduces the running example used throughout this paper. Section 3 introduces Inductive Logic Programming and the concrete algorithms used and in section 4 we apply ILP for automatically generating intentional pointcuts. Afterwards, we present the tools created to support our approach, compare with related work and state our conclusions.

2 Background and Problem Statement

2.1 Pattern-Based Pointcuts

The main problem in maintaining aspect-oriented code is the so-called *fragile pointcut problem* [21]. Pointcuts are deemed fragile when seemingly innocent changes to the base program, such as renaming or relocating a method, break a pointcut such that it no longer captures the joinpoints it is intended to capture. When code is added to a program and introduces new joinpoints in the joinpoint model of the program, pointcuts are similarly considered fragile when some of these new joinpoints should be captured by the pointcut but it fails to do so.

```
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 }
```

Fig. 1. A simple Point class

As described in our previous work [11] and that of others [20], pointcuts are particularly fragile when they are written in an enumerative style. As an

example take the Point class of figure 1. When adding an observer aspect, we need a pointcut that captures all executions of methods on the Point class that are state changing. A purely enumerative pointcut is shown in figure 2.

The pointcut language used in the examples here and the remainder of the paper is a pointcut language expressed using logic programming based on the CARMA pointcut language [11]. The main difference is that CARMA uses a fully dynamic joinpoint model, which for example allows conditions in pointcuts on the values associated with joinpoints. The pointcut language used in this paper has a purely static joinpoint model, which effectively equates joinpoints with shadow joinpoints. An extension of the work presented here that takes a dynamic joinpoint model into account is left for future work. The important point however is that the pointcut language used here similarly to CARMA also allows full access to the static joinpoint model of the program, which allows writing advanced intentional pointcuts.

```
1 stateChanges(?jpvar):
2     execution(?jpvar,setX).
3 stateChanges(?jpvar):
4     execution(?jpvar,setY).
```

Fig. 2. A pointcut for the Observer aspect, written in a purely enumerative style.

The pointcut of figure 2 matches if the joinpoint at hand is either the execution of method `setX` or the execution of method `setY`. Such an enumeratively described pointcut obviously breaks easily. For example, when we evolve the point class to a three-dimensional point and add a `setZ` method, the `stateChanges` pointcut does not match the added method and thus fails to comply with the intention of capturing all methods that change the state of a Point object.

The problem with enumerative pointcuts is of course the motivation for writing pointcuts in a more pattern-based style, exploiting a pattern that is exhibited by the joinpoints that should be captured. The pointcut in figure 3 uses *quantification* over the names of methods that start with `set`. It remains consistent when evolving the point to a three-dimensional point. However, consider for example the addition of a `reset` method that resets the x and y dimension of the point to the default values. This method does not have the *begins with the keyword set* pattern in common with the other state changing methods. Conversely, consider the addition of a method `setting` which simply returns the value of a setting, rather than doing any assignments. This method also exhibits the *begins with keyword set* pattern but should in fact not be captured by the pointcut. We can capture the `reset` and `setting` methods as a deviation from the pattern by including an extra condition that the name of the method may also be `reset` and should not be `setting`, but this tends to add an enumerative list of exceptions to the pointcut.

```
1 stateChanges(?jpvar):
2     execution(?jpvar,?methodName),
3     startsWith(?methodName,'set').
```

Fig. 3. A pointcut for the Observer aspect, written in a pattern-based style.

```
1 stateChanges(?jpvar):
2     execution(?jpvar,?methodName),
3     inMethod(?assignmentJP,?methodName),
4     isAssignment(?assignmentJP,?assignmentTarget),
5     instanceVariable(?assignmentTarget,?className),
```

Fig. 4. A pointcut for the observer

Using an advanced pointcut language that gives access to the full static joinpoint model of methods, it is possible to exploit a more robust pattern [11]. Figure 4 illustrates a pointcut that exploits the pattern that all the state changing methods contain an assignment to an instance variable of an object. This pointcut does not break when adding the `setting` or `reset` methods.

2.2 Automated Support for Pattern-Based Pointcuts

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 then too easily breaks when the program is evolved after refactoring.

In this paper we present the results of using a specific machine learning technique for deriving a pattern exhibited by examples. In particular we use *inductive logic programming*, which is in fact an algorithm that works similarly to the process we've described in the previous section for coming to an evolution-robust pattern-based pointcut. We further describe this relation informally in the next section, and present in detail the ILP algorithm.

3 Inductive Logic Programming

3.1 Logic Induction of Pointcuts

The algorithm of logic induction is similar to the process we followed in section 2.1 for coming to a more evolution-robust pattern-based pointcut. Informally, the way ILP works and the relationship to this manual process is as follows:

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

background information: A second 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 of inducing a logic rule for combinations of the positive examples. This is similar to the manual process we followed in the previous section: we take two examples such as the methods `setX` and `setY`, and find that in the background information the fact that the names of the methods start with `set` holds true.

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. This is similar to the process followed in the previous section where we added a `setting` method which should not be covered by the pointcut.

3.2 FOIL

In this paper we use the FOIL ILP algorithm [28]. FOIL learns hypotheses which are sets of first-order rules, similar to Horn clauses. However, since no literals containing function symbols are allowed, the rules are more restricted than Horn clauses. On the other hand, the rules are more expressive because literals appearing in the body of the rules may be negated.

Pseudo-code for the algorithm is shown in figure 5. The algorithm takes a top-down approach to ILP. Starting with the most general rule, FOIL specializes it until no more negative examples are covered. The algorithm involves a double loop to find suitable queries. In the outer loop the algorithm generates rules, each time starting with the most general rule, covering all examples. In the inner loop, it adds clauses to the rule, until no more negative examples are covered. The algorithm halts when all positive examples have been covered.

The algorithm generates candidate literals based on the literals and variables already present in the rule, and on predicates found in the background information. Suppose the current rule is $P(?x_1, ?x_2, \ldots, ?x_k) \leftarrow L_1 \ldots L_n$. FOIL now considers the following literals for addition as L_{n+1}.

- $Q(?v_1, \ldots, ?v_r)$, where Q is predicate occurring in the background information and where $?v_i (\forall i, 0 < i < r)$ is either a new variable or a variable already present in the rule. At least one of the variables $?v_i$ has to be present in rule.
- $Equal(?x_j, ?x_k)$, where $?x_j$ and $?x_k$ are variables already present in the rule.
- The negation of the literals formed in the rules above.

FOIL(*Target_predicate*, *Predicates*, *Examples*)

```
1: Pos ← Examples for which Target_predicate is true
2: Neg ← Examples for which Target_predicate is false
3: Learned_rules ← {}
4: while Pos is not empty do {learn a new rule}
5:    NewRule ← a new rule for Target_predicate with no preconditions
6:    NewRuleNeg ← Neg
7:    while NewRuleNeg is not empty do {specialize NewRule}
8:       Candidate_literals ← generate candidate new literals for NewRule
9:       calculate Foil_Gain for each literal in Candidate_literals
10:      add literal with highest Foil_Gain to preconditions of NewRule
11:      NewRuleNeg ← subset of NewRuleNeg satisfying NewRule preconditions
12:   end while
13:   Learned_Rules ← Learned_Rules ∪ {NewRule}
14:   Pos ← Pos\{ members of Pos covered by NewRule}
15: end while
16: return Learned_rules
```

Fig. 5. FOIL Algorithm

At each step of the inner loop a heuristic function is evaluated for all candidate literals. The result of this function shows how much the rule gains from adding this literal. The candidate literal which results in the highest gain is chosen as the next literal. This gain function, shown in figure 6, is a simple measure, based on the comparison of the number of covered positive (p) and negative (n) examples before (p_0, n_0) and after (p_1, n_1) the literal is added to the rule. The numbers of bindings that remain positive (t) after adding the literal to the rule is factored in.

$$Foil\_Gain(L, R) = t\left(\log_2 \frac{p_1}{p_1 + n_1} - \log_2 \frac{p_0}{p_0 + n_0}\right)$$

Fig. 6. Foil_Gain function

4 Applying ILP for Pointcut Abstraction

The FOIL algorithm is able to find rules from a set of logic facts. It requires a number of positive examples and a set of negative examples to avoid oversimplification. In addition, it expects a sufficiently large set of background information in order to be able to induce a rule. The positive examples for FOIL are the joinpoints where the aspect needs to be applied. They can either be manually selected

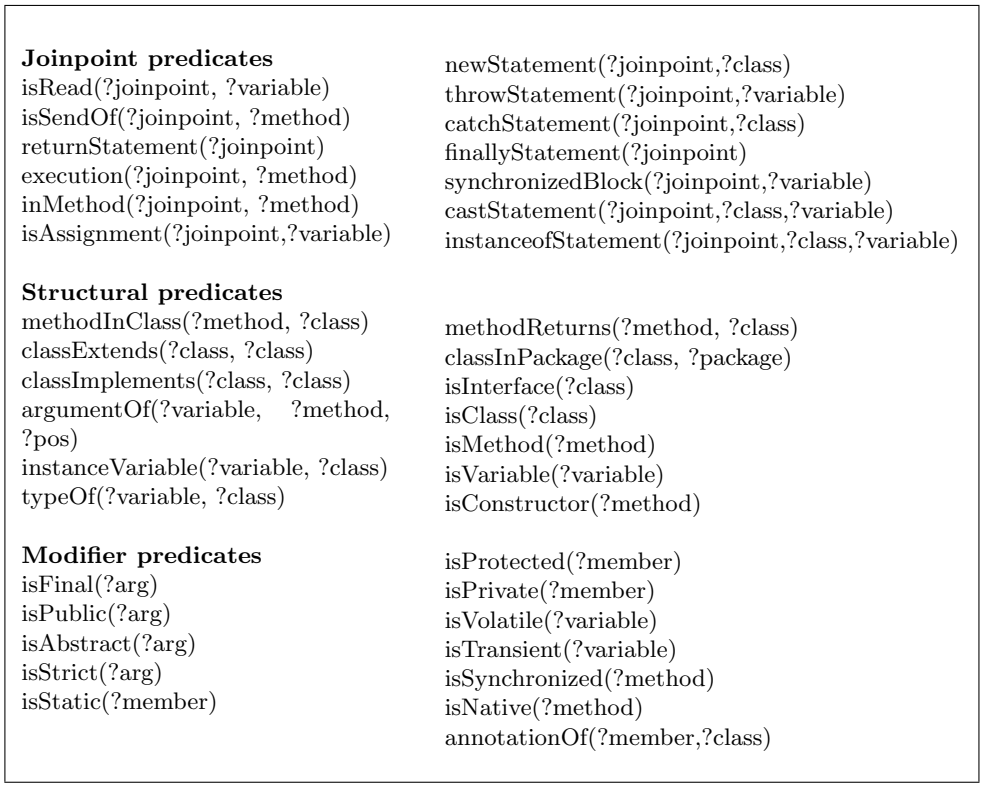

Fig. 7. Predicates available in the crosscut language to select joinpoints, the solutions for these predicates are used as background information for the ILP algorithm.

or automatically using for example an aspect mining technique. 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. These are the solutions of the predicates shown in figure 7, which also includes predicates about the relationships between classes etc. Because this pointcut language is based on a purely static joinpoint model, these solutions can be determined using only the program's source or compiled representation, i.e. compiled Java classes.

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. What we furthermore expect is that that the induced pointcut also is a non-fragile or robust pointcut. 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 we derived manually in section 2.1 as a benchmark for comparison.

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 (figure 7). This forms the background information for the logic induction algorithm, part of this generated background information is shown in figure 8.

```
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').
```

Fig. 8. Part of the background information for the Point class of figure 1.

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 `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(jp2)`. The resulting rule is shown in figure 9. 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.

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

Fig. 9. Induced stateChanges pointcut.

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.

Table 1. Generated facts statistics

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

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 10. In this case, the algorithm generates a pointcut that is sufficiently robust for evolution: it is in fact the same pointcut we determined manually in section 2.1.

4.2 Extended experiments

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

```
stateChanges(A):
    execution(A,B),
    inMethod(C,B),
    isAssignment(C,D),
    instanceVariable(D,E).
```

Fig. 10. Resulting pointcut when applying our approach to the AWT Point class.

Large fact database: We apply our approach to the complete Java AWT library in order to evaluate whether our approach 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 is the same as for the Java AWT Point class alone: the same pointcut as was determined manually in section 2.1 is induced. For a performance evaluation, we refer to section 5.

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 11, 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.

```
stateChanges(A):
    execution(A,B),
    inMethod(C,B),
    isAssignment(C,D),
    instanceVariable(D,E),
    not(isTransient(D)).
```

Fig. 11. Resulting pointcut for non-transient field assignments in Java AWT.

InEquality: The FOIL algorithm is also able to induce inequality for certain rule variables. For example, suppose we want to detect all methods that contain "illegal" assignments, namely assignments to instance variables of other classes.

The rule of figure 12 is induced when we apply this experiment to the AWT library. This rule declares that a method is illegally state changing when it contains an assignment to an instance variable that does not belong to the same class as the method.

```
1 illegalStateChanges(A):
2       execution(A,B),
3       methodInClass(B,C),
4       inMethod(D,B),
5       isAssignment(D,E),
6       instanceVariable(E,F),
7       C<>F.
```

Fig. 12. Resulting pointcut for field assignments from a different class than the class defining the field.

Recursion: Another advantage of the FOIL algorithm is its ability to induce recursive rules. For example, suppose we redefine state changing methods to also include execution joinpoints of methods that indirectly change the state of an object by invoking a method that is state changing. This is useful for implementations of the observer aspect that take into account the jumping aspect problem [4, 11]. In order to capture this pattern robustly, two pointcut rules are required, one of which is recursive. In this experiment our ILP implementation however did not induce such a recursive pointcut rule although theoretically the algorithm is able to induce recursive rules. The algorithm induces several rules that are unnecessarily complicated, depending on information that is irrelevant to the state changing concern. This pointcut breaks easily when the base program evolves because it is concerned with too much information. This failure is probably due to a problem in our current implementation of the algorithm. For instance, when we use method names as positive examples rather than joinpoints, a recursive rule is induced which does not exhibit such fragility issues, the resulting rule is shown in figure 13.

```
1 stateChanges(A):
2    inMethod(B,A),
3    isAssignment(B,C),
4    instanceVariable(C,D).
5 stateChanges(A):
6    inMethod(B,A),
7    isSendOf(B,C),
8    stateChanges(C).
```

Fig. 13. Recursive stateChanges rule.

5 Tool Support

Our approach is supported by a fully automatic tool-chain, which is illustrated in figure 14. The tool-chain consists of the following tools:

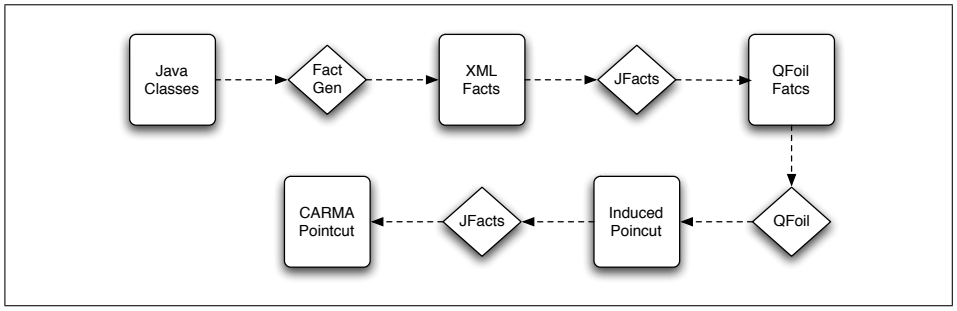

Fig. 14. The tool-chain for inducing CARMA pointcuts from Java classes.

- *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.
- *QFoil:* This tool is the implementation of the FOIL ILP algorithm by Ross Quinlan [29]. 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. This implementation of FOIL is particularly interesting because of its performance (see the benchmarks in the next paragraph).

In order to evaluate our approach performance-wise, we conduct several benchmark experiments with an increasingly large number of facts. The experiments were done using the state changing methods example. Table 2 shows the results[1]. In all cases, except for the toy Point class of course, the rule from Figure 10 was induced. The performance results are acceptable as the time required is not much more than compiling such a large set of classes. Considering the premature stage of the FactGen and JFacts tools, we believe that a significant improvement is still possible there.

[1] The timings were performed on an Intel Pentium 4 3Ghz. Each timing represents the average time of a single experiment, based on 100 experiments.

Table 2. Benchmark results of our prototype tool-chain.

	# classes	# facts	# joinpoints	FactGen+JFacts (s)	QFOIL (s)
Toy Point class	1	71	10	0.461	0.01
AWT Point class	1	364	70	0.5902	0.0142
25 classes from AWT	25	11622	2855	1.8098	0.8779
50 classes from AWT	50	42870	10982	3.9702	5.4671
75 classes from AWT	75	79403	21367	6.5163	4.4448
100 classes from AWT	100	88236	23409	7.1599	5.4526
AWT (no subpackages)	118	103752	27862	7.9929	7.1708

6 Related Work

To our knowledge, there exist no other approaches which try to automatically generate pattern-based pointcuts. In previous work [12] 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 [25], 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 [13, 24, 22, 15] 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 automaticity 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 [3]. Bruntink et al. [5] make use of clone detection techniques in order to isolate idiomatically implemented crosscutting concerns. These approaches are complementary with our approach in that the joinpoints they identify can

serve as positive examples for our ILP algorithm. 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 [14], FEAT [30], JQuery [17] and the Concern Manipulation Environment [16].

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 several successful experiments that apply our approach to a realistic and medium-scale case study. Although we present our approach using a logic pointcut language based on CARMA, there is no problem in applying the approach to other pointcut languages, e.g. AspectJ [18], as well.

Several facets are still open for improvement though:

- *Multiple Results:* Our current tools only generate one pointcut for a given set of joinpoints. In some cases, most notably when there is few background information (i.e. a small number of little classes), several alternative pointcuts are possible. Our current approach has a bias for short, non-negative and non-recursive rules. As we have described in the paper, this might not always lead to a (good) result. Therefore, it would be useful to allow presenting multiple pointcut results. To this end, we will need to experiment with alternative configurations of the FOIL algorithm in order to have several useful configurations.
- *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) [25] algorithm in an aspect mining context [12]. 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.
- *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 [10], cannot be induced. For this end, facts representing the run-time behavior of the program are necessary. We are currently investigating whether it is possible to induce such dynamic pointcuts using several program traces as background information.
- *Tool Integration:* Although our current tool works fully automatically, it is a stand-alone command-line tool that is not integrated in an IDE. We plan to develop an Eclipse plugin for our tool. This plugin can then be as a basis for inducing pattern-based pointcuts by other plugins which provide support for the refactoring process.

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. 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.
4. Johan Brichau, Wolfgang De Meuter, and Kris De Volder. Jumping aspects. In C. Lopes, L. Bergmans, M. D'Hondt, and P. Tarr, editors, *Workshop on Aspects and Dimensions of Concerns (ECOOP 2000)*, June 2000.
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. Bart De Win, Wouter Joosen, and Frank Piessens. Developing secure applications through aspect-oriented programming. pages 633–650. Addison-Wesley, Boston, 2005.
9. Maja D'Hondt and Viviane Jonckers. Hybrid aspects for weaving object-oriented functionality and rule-based knowledge. In Lieberherr [23], pages 132–140.
10. Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In Lieberherr [23], pages 141–150.
11. Kris Gybels and Johan Brichau. Arranging language features for pattern-based crosscuts. In Akşit [1], pages 60–69.
12. 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.
13. 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.
14. J. Hannemann. The Aspect Mining Tool web site. http://www.cs.ubc.ca/labs/spl/projects/amt.html.
15. 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.
16. 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.
17. Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In Akşit [1], pages 178–187.
18. 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.

19. 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.
20. Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *European Conference on Object-Oriented Programming, ECOOP 2005*, 2005.
21. 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.
22. Ramnivas Laddad. Aspect-oriented refactoring, dec 2003.
23. Karl Lieberherr, editor. *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*. ACM Press, March 2004.
24. Miguel Pessoa Monteiro. Catalogue of refactorings for aspectj. Technical Report UM-DI-GECSD-200401, Universidade Do Minho, 2004.
25. S. Muggleton and C. Feng. Efficient induction in logic programs. In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, 1992.
26. H. Ossher and P. Tarr. The shape of things to come: Using multi-dimensional separation of concerns with Hyper/J to (re)shape evolving software. *Comm. ACM*, 44(10):43–50, October 2001.
27. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(12):1053–1058, December 1972.
28. J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, August 1990.
29. Ross Quinlan. Qfoil: the reference foil implementation. Home page at `http://www.rulequest.com/Personal/`, 2005.
30. 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.
31. Wim Vanderperren, Davy Suvée, and Viviane Jonckers. Combining AOSD and CBSD in PacoSuite through invasive composition adapters and JAsCo. In *Net.ObjectDays 2003*, pages 36–50, September 2003.