Aspect Mining based on Control-Flow

Jens Krinke, Silvia Breu
FernUniversität in Hagen, Germany
krinke@acm.org, silvia.breu@fernuni-hagen.de

1 Introduction

Aspect mining tries to identify crosscutting concerns in existing systems and thus supports the adaption to an aspect-oriented design. This paper describes an automatic static aspect mining approach, where the control flow graphs of a program are investigated for recurring execution patterns based on different constraints, such as the requirement that the patterns have to exist in different calling contexts. A case study done with the implemented tool shows that most discovered crosscutting candidates are most often perfectly good style.

2 Aspect Mining

A major problem in re-engineering legacy code based on aspect-oriented principles is to find and to isolate these crosscutting concerns. This task is also called *aspect mining*. The detected concerns can be re-implemented as separate aspects, thereby improving maintainability and extensibility as well as reducing complexity. Aspect mining can also provide insights that enable us to classify common aspects which occur in different software systems, such as logging, timing, and communication.

Several approaches based on static program analysis techniques have been proposed for aspect mining. We have developed a dynamic program analysis approach [1] that mines aspects based on program traces. Based on the experience with the dynamic approach, we implemented a similar static analysis. This analysis extracts the execution relations from a control flow graph of the analyzed program. In particular, we immediately extract uniform and crosscutting execution relations without a previous step to extract unconstrained execution relations. However, the extraction is different for outside and inside execution relations. Here, we will only present inside-first ($R^{\in\top}$) and outside-before ($R^{\rightharpoonup}$) execution relations. Due to space constraints, we refer the reader to [1] for definitions and notations.

Inside-First Execution Relations. For these kind of execution relations, we extract the method invocations immediately following the entry of (invoked) methods from the control flow graph. Such a relation is uniform, if every path through the method starts with the same method call. Moreover, a possible simplification just considers the single-entry-single-exit regions starting at the methods' entry nodes. Such a relation $u \in_\top v$ means now that method u is the first method invocation inside the single-entry-single-exit region starting at the entry node of method v. The definition of crosscutting stays the same, thus u is a crosscutting method invocation if there are at least two uniform execution relations $u \in_\top v$ and $u \in_\top w$ ($v \neq w$).

Outside-Before Execution Relations. Here we extract all pairs of method invocations u, v if there exists a path from an invocation of method u to an invocation of method v without any method invocation in between. Such a pair is a uniform outside-before execution relation $u \rightharpoonup v$, if all paths from an invocation of method u contain an invocation of v as the next invocation. The first possible simplifications is to require that an invocation of u is post-dominated by an invocation of v without another invocation in between. The second simplifications is to require that any invocation of method u is followed by an invocation of v in all single-entry-single-exit regions containing an invocation of u.

3 Experiences

We have implemented the presented static mining on top of the Soot framework [2], which is used to compute the control flow graph of the analyzed program. Our tool traverses these control flow graphs and extracts the uniform and crosscutting inside-first and outside-before execution relations. As a first test case we have analyzed JHotDraw, version 5.4b1. For inside-first execution relations, the tool has identified 277 candidates with 1236 uniform and crosscutting relations, and for outside-before relations, 92 candidates with 294 relations.

It is interesting, that there are many more candidates for inside-first than for outside-before. Furthermore, there are a lot of candidates with just a small amount of crosscutting, e.g., 127 candidates that just crosscut two methods.

We will next discuss some of the identified candidates in detail. However, due to the large amount of identified candidates, we will only present the six largest candidates of each category.

Inside-First Relations The largest candidate consists of 49 uniform and crosscutting execution relations. The invoked method is "...*CollectionsFactory.current*". It is obvious that this is a method to access the current factory object, needed in many other methods of the system. This is clearly crosscutting, however, not a refactorable aspect.

The second largest candidate consists of 32 relations for the method "...*DrawingView.view*". This is again an acces-

sor method that returns the currently active view. Thus, it is crosscutting but not refactorable.

The same holds for the third and fourth candidate, which both consist of 24 relations. The relevant methods are "*...DecoratorFigure.getDecoratedFigure*" and "*...AbstractHandle.owner*" which are once again accessor methods.

For the fifth candidate, things are not different: It consists of 22 relations for the method "*...UndoadableAdapter.undo*" that checks whether the current object represents an undo-able action.

Things change for the sixth candidate consisting of 20 relations for method "*...AbstractFigure.willChange*". That method informs a figure that an operation will change the displayed content. This is the first candidate that is a crosscutting concern which could be refactored into an aspect.

Outside-Before Relations The largest discovered candidate consists of 13 uniform and crosscutting execution relations for the method "*...Iterator.next*". A closer look to the 13 invocations reveals that this crosscutting is more or less incidental: An operation is performed on the next element of a container.

The second largest candidate is somewhat interesting: It consists of 12 invocations before a call to "*...AbstractCommand.execute*", from which 11 are invocations of method "*createUndoActivity*". The other is an invocation of "*...ZoomDrawingView.zoomView*", which seems to be an *anomaly*. However, the other 11 invocations are of classes representing operations that change the figure and *zoomView* (probably) does not change it.

The next three largest candidates (consisting of 11, 9, and 8 relations) are again more or less incidental crosscutting concerns related to methods "*...DrawingView.drawing*", "*...List.add*", and "*...DrawingView.view*". However, it is interesting to see that *DrawingView.view* was also part of a large inside-first candidate.

Again, only the sixth largest candidate can be seen as crosscutting concern that can be refactored into an aspect. It consists of seven relations for method "*...AbstractFigure.willChange*". It is immediately called before methods that will change the displayed figure. However, it is interesting to see that this method has also appeared as an inside-first candidate, where the candidate is larger (20 relations).

A simple filter

We have seen in the last section that most of the discovered crosscutting concerns are not to be refactored, because they are perfectly valid in their characteristics. However, we want to identify crosscutting concerns that are more in the style of superimposition, i.e. that add behavior at the place where they are used but without having a direct dependence with the enclosing code.

A very simple, but very effective filter uses the signatures of the invoked methods. It is based on the assumption that any method that returns a value has been delegated a task to perform that is part of a bigger functionality/concern. This is like a trivial form of crosscutting that lead to the introduction of procedures and methods in programming languages. Thus, we assume that only void methods are not directly needed where they are invoked. Of course, this is over-simplifying because of reference parameters. The implemented filter extracts only those uniform and crosscutting execution relations that involve a void method.

size	relations	size	relations
2	30	9	0
3	15	10	1
4	11	11	1
5	1	13	1
6	1	17	1
7	2	20	1
8	2		

261 relations ($R^{\in\top}$) in 67 candidates

Table 1: Filtered Inside-First Execution Relations

A closer look at the extracted relations (see Table 1 for an overview) reveal that most of them have the characteristics of crosscutting concerns, especially the larger ones.

4 Conclusions

This initial evaluation of the static aspect mining tool has shown that most of the identified crosscutting candidates are not concerns refactorable into aspects. This is not much different from results in our previous dynamic aspect mining [1]. However, both approaches give interesting insights into the crosscutting behavior of the analyzed program. Moreover, as seen in the example for method *AbstractCommand.execute*, they can probably be used to discover *crosscutting anomalies*, an anomaly in the discovered execution relation pattern.

References

[1] Silvia Breu and Jens Krinke. Aspect mining using event traces. In *Proc. International Conference on Automated Software Engineering*, pages 310–315, 2004.

[2] Raja Vallee-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot – a java bytecode optimization framework. In *Proc. CASCON*, 1999.