Control-Flow-Graph-Based Aspect Mining

Jens Krinke
FernUniversität in Hagen, Germany
krinke@acm.org

Silvia Breu
NASA Ames Research Center, USA
silvia.breu@gmail.com

Abstract

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.

1. Introduction

The notion of *tangled code* refers to code that exists several times in a software system but cannot be encapsulated by separate modules using traditional module systems because it crosscuts the whole system. This makes software more difficult to maintain, to understand, and to extend. *Aspect-Oriented Programming* [4] provides new separation mechanisms for such complex *crosscutting concerns* [7].

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 [3, 5, 6, 10, 8, 2]. We have developed a dynamic program analysis approach [1] that mines aspects based on program traces. During program execution, program traces are generated, which reflect the run-time behavior of a software system. These traces are then investigated for recurring execution patterns. Different constraints specify when an execution pattern is "recurring". These include the requirement that the patterns have to exist in different calling contexts in the program trace. The dynamic analysis approach monitors actual (i.e., run-time) program behavior instead of potential behavior, as static program analysis does. To explore the differences between static and dynamic analyses in aspect mining, we have started to develop a static analysis variant of our approach. From early results we experienced two things:

- The results of the static and dynamic analysis are different due to various reasons.
- Crosscutting concerns are often perfectly good style, because they result from delegation and coding style guides.

The first point is obvious and thus, only the second point will be discussed in the following. The next section contains an introduction to our dynamic aspect mining approach. A static aspect mining approach based on the dynamic variant is presented in Section 3. Section 4 contains a case study, Section 5 discusses the results and concludes, before Section 6 discusses related work.

2. Dynamic Aspect Mining

The basic idea behind dynamic analysis algorithms is to observe run-time behavior of software systems and to extract information from the execution of the programs. The dynamic aspect mining approach introduced here is based on the analysis of program traces which mirror a system's behavior in certain program runs. Within these program traces we identify recurring execution patterns which describe certain behavioral aspects of the software system. We expect that recurring execution patterns are potential crosscutting concerns which describe recurring functionality in the program and thus are possible aspects.

In order to detect these recurring patterns in the program traces, a classification of possible pattern forms is required. Therefore, we introduce so-called *execution relations*. They describe in which relation two method executions are in the program trace.

2.1. Classification of Execution Relations

The definition of execution relations in our analysis approach is based on program traces. Intuitively, a program trace is a sequence of method invocations and exits. We only consider entries into and exits from method executions because we can then easily keep track of the relative order in which method executions are started and finished. We focus on method executions because we want to analyze object-oriented systems where logically related functionality is encapsulated in methods. Formally, a *program trace* T_P of a program P with method signatures $\mathcal{N}_P$ is defined as a list $[t_1, \ldots, t_n]$ of pairs $t_i \in (\mathcal{N}_P \times \{ent, ext\})$, where ent marks entering a method execution, and ext marks exiting a method execution.

To make the program traces easier to read, the ent- and ext-points are represented by { and } respectively, and the redundant $name$-information is discarded from the ext-points as the trace structure implies to which $name$ the ext belongs. Figure 1 shows an example trace.

```
 1 B() {
 2   C() {
 3     G() {}
 4     H() {}
 5   }
 6 }
 7 A() {}
 8 B() {
 9   C() {}
10 }
11 A() {}
12 B() {
13   C() {
14     G() {}
15     H() {}
16   }
17   J() {}
18 }
19 F() {
20   K() {}
21   I() {}
22 }
23 J() {}
24 G() {}
25 H() {}
26 A() {}
27 B() {
28   C() {}
29   G() {}
30   F() {
31     K() {}
32     I() {}
33   }
34 }
35 D() {
36   C() {}
37   A() {}
38   B() {
39     C() {}
40   }
41   K() {}
42   I() {
43     J() {}
44   }
45   G() {}
46   E() {}
47 }
```

Figure 1. Example trace

Crosscutting concerns are now reflected by the two different *execution relations* that can be found in program traces: A method can be executed either after the preceding method execution is terminated (e.g., `H()` in line *4* is executed after `G()` in line *3*), or inside the execution of the preceding method call (e.g., `C()` in line *2* is executed inside `B()` in line *1*). We distinguish between these two cases and say that there are outside- and inside-execution relations in program traces. However, this distinction alone is not yet sufficient for aspect mining. For example, the execution of `B()` in line *27* has three methods executed inside its execution, `C()`, `G()`, and `F()` in lines *28* ff., but the information which of those methods comes first is lost. We thus define formally:

$u \rightharpoonup v,\ u, v \in \mathcal{N}_P$, is called an *outside-before-execution relation* if $[(u, ext), (v, ent)]$ is a sublist of T_P. $S^{\rightharpoonup}(T_P)$ is the set of all outside-before-execution relations in a program trace T_P. This relation can also be reversed, i.e., $v \leftharpoonup u$ is an *outside-after-execution relation* if $u \rightharpoonup v \in S^{\rightharpoonup}(T_P)$. The set of all outside-after-execution relations in a program trace T_P is then denoted with $S^{\leftharpoonup}(T_P)$.

$u \in_\top v,\ u, v \in \mathcal{N}_P$ is called an *inside-first-execution relation* if $[(v, ent), (u, ent)]$ is a sublist of T_P. $u \in_\bot v$ is called an *inside-last-execution relation* if $[(u, ext), (v, ext)]$ is a sublist of T_P. $S^{\in_\top}(T_P)$ is the set of all inside-first-execution relations in a program trace T_P, $S^{\in_\bot}(T_P)$ is the set of all inside-last-execution relations. In the following, we drop T_P when it is clear from the context.

For the example trace shown in Figure 1 we thus get the following set $S^{\rightharpoonup}$ of outside-before-execution relations:

$$\begin{aligned} S^{\rightharpoonup} = \{ & \texttt{B()} \rightharpoonup \texttt{A()}, \texttt{G()} \rightharpoonup \texttt{H()}, \texttt{A()} \rightharpoonup \texttt{B()}, \texttt{C()} \rightharpoonup \texttt{J()}, \\ & \texttt{B()} \rightharpoonup \texttt{F()}, \texttt{K()} \rightharpoonup \texttt{I()}, \texttt{F()} \rightharpoonup \texttt{J()}, \texttt{J()} \rightharpoonup \texttt{G()}, \\ & \texttt{H()} \rightharpoonup \texttt{A()}, \texttt{B()} \rightharpoonup \texttt{D()}, \texttt{C()} \rightharpoonup \texttt{G()}, \texttt{G()} \rightharpoonup \texttt{F()}, \\ & \texttt{C()} \rightharpoonup \texttt{A()}, \texttt{B()} \rightharpoonup \texttt{K()}, \texttt{I()} \rightharpoonup \texttt{G()}, \texttt{G()} \rightharpoonup \texttt{E()} \} \end{aligned}$$

The set $S^{\leftharpoonup}$ of outside-after-execution relations can be found directly in the trace or simply by reversing $S^{\rightharpoonup}$. The sets $S^{\in_\top}$ of inside-first-execution relations and $S^{\in_\bot}$ of inside-last-execution relations are as follows:

$$\begin{aligned} S^{\in_\top} = \{ & \texttt{C()} \in_\top \texttt{B()}, \texttt{G()} \in_\top \texttt{C()}, \texttt{K()} \in_\top \texttt{F()}, \texttt{C()} \in_\top \texttt{D()}, \\ & \texttt{J()} \in_\top \texttt{I()} \} \\ S^{\in_\bot} = \{ & \texttt{H()} \in_\bot \texttt{C()}, \texttt{C()} \in_\bot \texttt{B()}, \texttt{J()} \in_\bot \texttt{B()}, \texttt{I()} \in_\bot \texttt{F()}, \\ & \texttt{F()} \in_\bot \texttt{B()}, \texttt{J()} \in_\bot \texttt{I()}, \texttt{E()} \in_\bot \texttt{D()} \} \end{aligned}$$

2.2. Execution Relation Constraints

Recurring execution relations in the program traces can be seen as indicators for more general execution patterns. To decide under which circumstances certain execution relations are recurring patterns in traces and thus potential crosscutting concerns in a system, constraints have to be defined. The constraints will implicitly also formalize what crosscutting means.

However, for technical reasons we have to encode that there is no further method execution between nested method executions or between method invocation and method exit. This absence of method executions is represented by the designated empty method signature ϵ. Therefore, the definition of execution relations is extended such that each sublist of a program trace T_P induces not only relations defined above but also additional relations involving ϵ. Table 1 summarizes this conservative extension. It shows for each two-element sublist of the trace (on the left side) the execution relations that follow from that sublist (on the right side). The execution relations added by the introduction of ϵ are annotated with an asterisk ($*$).

The program trace remains as defined before with method signatures from $\mathcal{N}_P$ whereas the execution relations now can consist of method signatures from $\mathcal{N}_P \cup \{\epsilon\}$.

Trace-sublist $(\mathcal{N}_P)$		**Relation** s $(\mathcal{N}_P \cup \{\epsilon\})$
(u, ext)	(v, ent)	$u \rightharpoonup v$, $v \leftharpoondown u$
(v, ent)	(u, ent)	$\epsilon \rightharpoonup u^*$, $u \leftharpoondown \epsilon^*$, $u \in_\top v$
BOL	(u, ent)	$\epsilon \rightharpoonup u^*$, $u \leftharpoondown \epsilon^*$, $u \in_\top \epsilon^*$
(u, ext)	(v, ext)	$u \rightharpoonup \epsilon^*$, $\epsilon \leftharpoondown u^*$, $u \in_\perp v$
(u, ext)	EOL	$u \rightharpoonup \epsilon^*$, $\epsilon \leftharpoondown u^*$, $u \in_\perp \epsilon^*$
(w, ent)	(w, ext)	$\epsilon \in_\top w^*$, $\epsilon \in_\perp w^*$

BOL/EOL denote begin/end of list

Table 1. Extended execution relations

Thus, the sets $S^{\rightharpoonup}$, $S^{\leftharpoondown}$, $S^{\in_\top}$, and $S^{\in_\perp}$ also include execution relations involving ϵ. Now, we can define the constraints for the dynamic analysis.

Formally, an execution relation $s = u \circ v \in S^\circ$, $\circ \in \{\rightharpoonup, \leftharpoondown, \in_\top, \in_\perp\}$, is called *uniform* if $\forall w \circ v \in S^\circ : u = w, u, v, w \in \mathcal{N}_P \cup \{\epsilon\}$ holds, i.e., it exists in always the same composition. $\widehat{U}^\circ$ is the set of execution relations $s \in S^\circ$ which satisfy this requirement. This constraint is easy to explain. Consider an outside-before-execution relation $u \rightharpoonup v$. This is defined as recurring pattern if each execution of v is preceded by an execution of u. The argumentation for outside-after-execution relations is analogous. The uniformity-constraint also applies to inside-execution relations. An inside-execution relation $u \in_\top v$ (or $u \in_\perp v$) can only be a recurring pattern in the given program trace if v never executes another method than u as first (or last) method inside its body.

We now drop the ϵ-relations and define a further analysis constraint: An execution relation $s = u \circ v \in U^\circ = \widehat{U}^\circ \setminus \{u \circ v \mid u = \epsilon \vee v = \epsilon\}$ is called *crosscutting* if $\exists s' = u \circ w \in U^\circ : w \neq v,\ u, v, w \in \mathcal{N}_P$ holds, i.e., it occurs in more than a single calling context in the program trace T_P. For inside-execution relations $u \in_\top v$ (or $u \in_\perp v$) the calling context is the surrounding method execution v. For outside-execution relations $u \rightharpoonup v$ (or $u \leftharpoondown v$) the calling context is the method v invoked before (or after) which always method u is executed. R° is the set of execution relations $s \in U^\circ$ which satisfy this requirement. Execution relations $s \in R^\circ$ are also called *aspect candidates* as they represent the potential crosscutting concerns of the analyzed software system.

2.3. Aspect Mining Algorithm

The constraints described above can be implemented by a relatively straightforward algorithm to actually compute the sets R° of uniform, crosscutting execution relations that represent the aspect candidates. In our running example, uniformity narrows down the potential aspect candidates to the following sets of execution relations:

$$\begin{aligned} U^{\rightharpoonup} &= \{\texttt{B()} \rightharpoonup \texttt{D()}, \texttt{G()} \rightharpoonup \texttt{E()}, \texttt{G()} \rightharpoonup \texttt{H()}, \texttt{K()} \rightharpoonup \texttt{I()}\} \\ U^{\leftharpoondown} &= \{\texttt{B()} \leftharpoondown \texttt{A()}, \texttt{I()} \leftharpoondown \texttt{K()}\} \\ U^{\in_\top} &= \{\texttt{C()} \in_\top \texttt{B()}, \texttt{C()} \in_\top \texttt{D()}, \texttt{K()} \in_\top \texttt{F()}\} \\ U^{\in_\perp} &= \{\texttt{E()} \in_\perp \texttt{D()}, \texttt{I()} \in_\perp \texttt{F()}\} \end{aligned}$$

After we enforce the crosscutting constraint, we obtain the final sets R° of aspect candidates which comply with uniformity *and* crosscutting.

$$\begin{aligned} R^{\rightharpoonup} &= \{\texttt{G()} \rightharpoonup \texttt{H()}, \texttt{G()} \rightharpoonup \texttt{E()}\},\ R^{\leftharpoondown} = \varnothing \\ R^{\in_\top} &= \{\texttt{C()} \in_\top \texttt{B()}, \texttt{C()} \in_\top \texttt{D()}\},\ R^{\in_\perp} = \varnothing \end{aligned}$$

3. Static Aspect Mining

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.

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.

4. Experiences

We have implemented the presented static mining on top of the Soot framework [9], which is used to compute the

size	relations	size	relations
2	127	13	4
3	55	15	2
4	30	16	1
5	12	17	2
6	9	18	1
7	7	19	1
8	7	20	1
9	3	22	1
10	3	24	2
11	3	32	1
12	4	49	1

1236 relations ($R^{\in\top}$) in 277 candidates

Table 2. Inside-First Execution Relations

size	relations	size	relations
2	53	8	1
3	19	9	1
4	4	11	1
5	6	12	1
6	3	13	1
7	2		

294 relations ($R^{\rightharpoonup}$) in 92 candidates

Table 3. Outside-Before Execution Relations

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. Tables 2 and 3 show the results. 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.

4.1. 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 accessor 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 candidates 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.

4.2. 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 12 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).

5. Discussion, Conclusions, and Future Work

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.

These results are preliminary because of the small amount of analyzed candidates (12) in a single test program. However, based on the previous results from the dynamic approach, our hypothesis is that the results will not change and are general. This would mean that aspect mining will have hard times to identify candidates that are really refactorable into aspects. Therefore, future work will continue in three directions:

1. A large-scale analysis of discovered candidates for a large set of programs with static and dynamic analysis.
2. Development of a filter which extracts the refactorable candidates from the discovered candidates.
3. A comparison with other aspect mining approaches.

6. Related Work

There only exists a small set of automatic aspect mining approaches. In most approaches one has to specify a pattern that can be searched for in the source code [3, 10].

Tourwe [8] uses concept analysis to identify aspectual views in programs. The extraction of elements and attributes from the names of classes, methods, and variables, formal concept analysis is used to group those elements into concepts that can be seen as aspect candidates.

Some other approaches rely on clone detection techniques to detect tangled code in the form of crosscutting concerns:

Bruntink [2] evaluated the use of those clone detection techniques to identify crosscutting concerns. Their evaluation has shown that some of the typical aspects are discovered very well while some are not.

Ophir [6] identifies initial re-factoring candidates using a control-based comparison. The initial identification phase builds upon code clone detection using program dependence graphs. The next step filters undesirable re-factoring candidates. It looks for similar data dependencies in subgraphs representing code clones. The last phase identifies similar candidates and coalesces them into sets of similar candidates, which are the re-factoring candidate classes.

References

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

[2] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwe. An evaluation of clone detection techniques for identifying cross-cutting concerns. In *Proc. International Conference on Software Maintenance*, 2004.

[3] W. G. Griswold, Y. Kato, and J. J. Yuan. Aspect Browser: Tool Support for Managing Dispersed Aspects. Technical Report CS99-0640, Department of Computer Science and Engineering, UC, San Diego, 1999.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *European Conf. on Object-Oriented Programming (ECOOP)*, 1997.

[5] N. Loughran and A. Rashid. Mining Aspects. In *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (AOSD Sat. workshop)*, 2002.

[6] D. Shepherd and L. Pollock. Ophir: A Framework for Automatic Mining and Refactoring of Aspects. Technical Report 2004-03, University of Delaware, 2003.

[7] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *21st Intl. Conf. on Software Engineering (ICSE)*, pages 107–119, 1999.

[8] T. Tourwe and K. Mens. Mining aspectual views using formal concept analysis. In *Proc. IEEE International Workshop on Source Code Analysis and Manipulation*, 2004.

[9] R. Vallee-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot – a java bytecode optimization framework. In *Proc. CASCON*, 1999.

[10] C. Zhang and H.-A. Jacobsen. Quantifying Aspects in Middleware Platforms. In *2nd Intl. Conf. on Aspect-Oriented Software Development (AOSD)*, pages 130–139, 2003.