Mining Control Flow Graphs for Crosscutting Concerns

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

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 instances of crosscutting delegation and should not be refactored into aspects.

1. Introduction

The notion of *scattered 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* [11] provides new separation mechanisms for such complex *crosscutting concerns* [19].

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. It is still an ongoing discussion what types of crosscutting concerns should be refactored into aspects to improve the quality of a system. It is our belief that only superimposed crosscutting behavior should be refactored, i.e. behavior that is not core functionality of a system. Others believe that aspects can be used to implement almost anything. However, there is not yet enough experience with aspect oriented programming to judge whether this is true. Independent of this, identification of the crosscutting concerns in a system is still very useful for the system comprehension.

Several approaches based on static program analysis techniques have been proposed for aspect mining (see related work in Section 4 for a discussion). We have previously developed a dynamic program analysis approach [2] 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., runtime) program behavior instead of potential behavior, as static program analysis does. Because it is not always possible to execute the program that should be analyzed and to explore the differences between static and dynamic analyses in aspect mining, we have developed a static analysis variant of our approach [12] that analyzes control flow graphs for recurring execution patterns. This work will show how the execution relations and their constraints can be generalized, it will present an algorithm to compute execution relations from control flow graphs, and will present an in-depth case study of the identified relations in JHotDraw. We experienced two main things with this approaches:

- 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 (issues are not executed code, late binding, etc.) and thus, only the second point will be discussed in the following. The evaluation of the results of the static aspect mining approach will specifically try to distinguish crosscutting concerns that are instances of delegation from those that can be regarded as superimposed behavior. This distinction helps us to classify whether a refactoring of the detected crosscutting concern into an aspect will im-

prove the systems quality (in the sense of maintainability, complexity, comprehensibility).

The next section contains a description of our static aspect mining approach based on control flow graphs. Section 3 contains a case study, followed by related work. Section 5 discusses the results and concludes.

2. Aspect Mining based on Execution Relations

The dynamic aspect mining approach previously introduced [2] 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 have seen 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 based on so-called *execution relations* has been introduced. They describe in which relation two method executions are in the program trace. However, often enough a program that should be analyzed cannot be executed. For such circumstances we have developed a static analysis that extracts the execution relations from a control flow graph. The execution relations are a general concept and in the following it is shown how the execution relations and their constraints are defined for this static approach.

A *control flow graph* (CFG) is a directed attributed graph $G = (N, E, n^s, n^e)$ with node set N and edge set E. The statements and predicates are represented by nodes $n \in N$ and the control flow between statements is represented by *control flow edges* $(n, m) \in E$, written as $n \rightarrow m$. E contains control flow edge e, iff the statement represented by node $\mathsf{target}(e)$ may immediately be executed after the statement represented by $\mathsf{source}(e)$, i.e. no other statement is executed in between. Two special nodes $n^s \in N$ and $n^e \in N$ are distinguished, the START node n^s and the EXIT node n^e, which represent beginning and end of the program. Node n^s does not have predecessors and node n^e does not have successors. Each procedure or method $p \in \mathcal{P}$ of a program is represented with its own control flow graph $G_p = (N_p, E_p, n^s_p, n^e_p)$, where $\forall p, q : p \neq q \Rightarrow N_p \cap N_q = \emptyset \wedge E_p \cap E_q = \emptyset$ and $N^\star = \bigcup_p N_p$, $E^\star = \bigcup_p E_p$ represent the set of nodes and edges for the complete program that is represented by the graph $G^\star = (N^\star, E^\star)$. Note that we assume a unique EXIT node which is a join in presence of multiple `return` statements. In languages like Java there may be multiple exits due to exception handling. However, in our approach such exits are ignored and execution relations are only identified in the parts of the source code that lead to a normal exit.

```
void m1() {
  a();
  b();
  a();
}
void m2() {
  a();
  if (...) {
    b();
    a();
  }
}
void m3() {
  a();
  c();
}
```

Figure 1. Example

Usually, the procedures' graphs are connected with edges that represent procedure or method calls, however, such edges are not important for this paper and we assume that the graphs are not connected. In the following we will only use the term method to denote methods or procedures.

2.1. Classification of Execution Relations

In the control flow graphs we focus on method calls (and executions) because we want to analyze imperative or object-oriented systems where logically related functionality is encapsulated in methods.

Crosscutting concerns are reflected by the two different *execution relations* that can be found in control flow graphs: A method can be executed either after the preceding method execution is terminated or inside the execution of another method call. We distinguish between these two cases and say that there are *outside-* and *inside-execution relations* in control flow graphs. We thus define formally:

$u \rightharpoonup v$ with $u, v \in \mathcal{P}$, is called an *outside-before-execution relation* if there is a path $n_u \rightarrow^\star n_v$ in $G^\star$ where n_u is a call of u, n_v is a call of v and there is no other call on the path. This is read as "u is executed before v". $S^{\rightharpoonup}(G)$ is the set of all outside-before-execution relations in a call graph G. This relation can also be reversed, i.e., $v \leftharpoonup u$ is an *outside-after-execution relation* if $u \rightharpoonup v \in S^{\rightharpoonup}(G)$. The relation $u \leftharpoonup v$ can be read as "v is executed after u". The set of all outside-after-execution relations in a graph G is then denoted with $S^{\leftharpoonup}(G)$. Figure 1 shows a small fragment that will be used as an example. The following relations can be identified: `a` $\rightharpoonup$ `b`, `b` $\rightharpoonup$ `a`, `a` $\rightharpoonup$ `c`, `b` $\leftharpoonup$ `a`, `a` $\leftharpoonup$ `b` and `c` $\leftharpoonup$ `a`.

$u \in_\top v$ with $u, v \in \mathcal{P}$ is called an *inside-first-execution relation* if there is a path $n_v \rightarrow^\star n_u$ in G_u such that $n_v = n_v^s$ is the START node, n_u is a call of u and there is no other call on the path. The relation $u \in_\top v$ can be read as "u is executed first in v". $u \in_\bot v$ is called an *inside-last-execution relation* if if there is a path $n_u \rightarrow^\star n_v$ in G_u such that $n_v = n_v^s$ is the EXIT node, n_u is a call of u and there is no other call on the path. The relation $u \in_\bot v$ can be read as "u is executed last in v". $S^{\in_\top}(G)$ is the set of all inside-first-execution relations in a control flow graph G, $S^{\in_\bot}(G)$ is the set of all inside-last-execution relations. In the following, we drop G when it is clear from the context. In Figure 1, the following relations can be identified: $\texttt{a} \in_\top \texttt{m1}$, $\texttt{a} \in_\top \texttt{m2}$, $\texttt{a} \in_\top \texttt{m3}$, $\texttt{a} \in_\bot \texttt{m1}$, $\texttt{a} \in_\bot \texttt{m2}$ and $\texttt{c} \in_\bot \texttt{m3}$.

There is one special case that has to be represented explicitly. Whenever there is a path $n_p^s \rightarrow^\star n_p^e$ that does not contain any call, the following two relations are explicitly generated: $\epsilon \in_\top p$ and $\epsilon \in_\bot p$. These two relations capture the possibility that no call to a method occurs during the execution of method p. Moreover, any inside-first- or inside-last-execution relation also generates similar constraints on the outside execution relations. An inside-first-execution relation $u \in_\top v$ generates $\epsilon \rightharpoonup u$ as there is no other call before u is called and an inside-last-execution relation $u \in_\bot v$ generates $\epsilon \leftharpoonup u$ because there is no other call after u is called. These generated relations are called *epsilon relations*. In Figure 1, the following epsilon relations can be identified: $\epsilon \rightharpoonup \texttt{a}$, $\epsilon \leftharpoonup \texttt{a}$, and $\epsilon \leftharpoonup \texttt{c}$.

The above execution relations including the special cases are similar to the execution relations defined for the dynamic aspect mining approach and most of the following constraints can now be used in both approaches.

2.2. Execution Relation Constraints

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

Formally, an execution relation $s = u \circ v \in S^\circ$, $\circ \in \{\rightharpoonup, \leftharpoonup, \in_\top, \in_\bot\}$, is called *uniform* if $\forall w \circ v \in S^\circ : u = w$ with $u, v, w \in \mathcal{P} \cup \{\epsilon\}$ 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 every 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 $u \in_\bot v$) can only be a recurring pattern if v never executes another method than u as first (or last) method inside its body. In Figure 1, only the following relations are uniform: $\texttt{a} \rightharpoonup \texttt{b}$, $\texttt{a} \rightharpoonup \texttt{c}$, $\texttt{a} \leftharpoonup \texttt{b}$, $\texttt{a} \in_\top \texttt{m1}$, $\texttt{a} \in_\top \texttt{m2}$, $\texttt{a} \in_\top \texttt{m3}$, $\texttt{a} \in_\bot \texttt{m1}$, $\texttt{a} \in_\bot \texttt{m2}$, $\texttt{c} \in_\bot \texttt{m3}$, and $\epsilon \leftharpoonup \texttt{c}$.

We now drop the ϵ-relations and define a further analysis constraint: An execution relation $s = u \circ v \in U^\circ = \widehat{U}^\circ \backslash \{u \circ v \mid u = \epsilon \vee v = \epsilon\}$ is called *crosscutting* if $\exists s' = u \circ w \in U^\circ : w \neq v$ with $u, v, w \in \mathcal{P}$ holds, i.e., it occurs in more than a single calling context in the control flow graph. For inside-execution relations $u \in_\top v$ (or $u \in_\bot v$) the calling context is the surrounding method execution v. For outside-execution relations $u \rightharpoonup v$ (or $u \leftharpoonup v$) the calling context is the method v invoked before (or after) method u. 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. In Figure 1, only the following relations are uniform and crosscutting: $\texttt{a} \rightharpoonup \texttt{b}$, $\texttt{a} \rightharpoonup \texttt{c}$, $\texttt{a} \in_\top \texttt{m1}$, $\texttt{a} \in_\top \texttt{m2}$, $\texttt{a} \in_\top \texttt{m3}$, $\texttt{a} \in_\bot \texttt{m1}$, and $\texttt{a} \in_\bot \texttt{m2}$.

The above definitions are exactly the same as in the dynamic approach and the technique to extract the uniform and crosscutting relations (which is not presented here) has been reused in the static approach that is presented next.

2.3. Static Aspect Mining

We implemented a tool to extract and evaluate the execution relations from the previous section. This analysis extracts the execution relations from a control flow graph of the analyzed program. In particular, we generate the control flow graphs for the methods of a program to analyze and the traverse the edges. The algorithm itself is basically a reaching definitions data flow analysis, where the definitions are replaced by the calls occurring at the nodes of the control flow graph. As this is well known data flow analysis, no details are given here. Only the data flow equations are given in the following.

Let $C \subseteq N^\star$ be the set of nodes that are calls to a method and let $c(n)$ be the called method of a node $n \in C$. A call of a method p at a node n ($p = c(n)$) *reaches* a (not necessarily different) node m, if a path $P = \langle n_1, \ldots, n_k \rangle$ in G exists, such that

1. $k > 1$
2. $n_1 = n \wedge n_k = m$
3. $\forall 1 < i < k : n_i \notin C$

To obey the epsilon relation, START and EXIT nodes are assumed to be call node with $\forall p \in \mathcal{P} : n_p^s \in C \wedge n_p^e \in C \wedge c(n_p^s) = \epsilon \wedge c(n_p^e) = \epsilon$.

Now $RC(n)$ is defined as the set of reaching calls at node n. They can be computed via a data flow analysis framework: Therefore, the local abstract semantics are described

with a transfer function over the set of calls x that reach a node n:

$$[\![n]\!](x) = \big(x - \mathsf{kill}(n)\big) \cup \mathsf{gen}(n)$$

This equation means that calls reaching the entry of a node n which are not killed by n, reach the exit of n together with calls generated by n. In the case of reaching calls, the sets kill and gen are easily defined:

$$\mathsf{gen}(n) = \begin{cases} \{p \mid p = c(n)\} & \text{if } n \in C \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathsf{kill}(n) = \begin{cases} \mathcal{P} & \text{if } n \in C \\ \emptyset & \text{otherwise} \end{cases}$$

A call reaches a node n, if a path from the START node n^s to n exists, on which the call is not killed. At the START node, only the ϵ call is available. If there exists more than one path, the reaching definitions of all paths are merged (by union in this case):

$$RC(n) = \bigcup_{p=\langle s,\ldots,n\rangle} [\![p]\!](\{\epsilon\})$$

This is an instance of a *meet-over-all-paths (MOP)* solution. In presence of loops there are infinite paths, which make the computation of the MOP solution impossible. Therefore, only the *minimal-fixed-point (MFP)* solution is computed:

$$RC(n) = \begin{cases} \{\epsilon\} & n = n_p^s \\ [\![n]\!](\bigcup_{m \to n} RC(m)) \end{cases}$$

Because of the properties of the transfer function the minimal-fixed-point solution is equal to the meet-over-all-paths solution.

From the reaching calls sets the four execution relations can be generated:

- $\forall n \in C \wedge m \in RC(n) : m \rightharpoonup n \wedge n \leftharpoonup m$
  Any call that reaches another call causes an outside-before-execution and an outside-after-execution relation between the two calls. (The epsilon relations are automatically generated.)
- $\forall n \in C \wedge n \in N_p \wedge \epsilon \in RC(n) : n \in_\top p$
  If the ϵ call reaches another call, then there exists a path from the START node to the call without another call and the corresponding inside-first-execution relation is generated. (The epsilon relations are generated if the ϵ call reaches the EXIT node.)
- $\forall n_p^e \in C \wedge m \in RC(n_p^e) : m \in_\bot p$
  Any call that reaches the EXIT node generates a corresponding inside-last-execution relation. (The epsilon relation is generated if the ϵ call reaches the EXIT node.)

size	candidates	size	candidates
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 1. Inside-First Execution Relations

3. Experiences

We have implemented the presented static mining on top of the Soot framework [22], 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 test case we have analyzed JHotDraw, version 5.4b1, which is a well known system and has often been used in evaluations of aspect mining techniques. It is a drawing application and demonstrates good use of design patterns. It contains about 18.000 lines of source code and 2.900 methods. Moreover, it has been extensively analyzed by Marin et al. [14] and we will use their results, which are available at a website[1], for comparison.

In the following, we will concentrate on inside-first and outside-before relations. The reasons are that the results for outside-after relations are very similar to the outside-before relations and because of the representation of exception handling in Soot, the results for inside-last relations are corrupted. Tables 1 and 2 show the results. The first column shows the size of detected crosscutting candidates measured by the number of crosscutted methods (the number of different methods v for a unique method u with relation $u \circ v \in R^\circ$). The second column shows the number of candidates of the size. 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.

1 http://swerl.tudelft.nl/amr/

size	candidates	size	candidates
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 2. Outside-Before Execution Relations

We will next discuss some of the identified candidates in detail ([12] presented only preliminary results). However, due to the large number of identified candidates, we will only present the six largest candidates of each category first. This initial discussion is used to argue that many of the identified crosscutting concerns are the result of delegations and should not be refactored into aspects. Instead, a filter is needed that removes the delegations from the results. Such a filter is presented after the first discussion with a detailed presentation of the then filtered results.

3.1. 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. This has also been observed by others, for example Marin et al [14] classified this method as a utility method which is ignored in their fan-in analysis.

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. (Again, this can be regarded as a 'utility' method.)

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. It is interesting to note that Marin et al. classified the owner method as part of the Observer crosscutting concern. However, we have decided to regard this as an accessor method.

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. However, this method call belongs to a well known crosscutting concern in JHotDraw, the Undo crosscutting concern which has been refactored by Marin [13]. This refactoring is complicated and not only related to the undo method. As our goal is to identify simple crosscutting concerns, this method is still classified as not refactorable (with small effort).

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. This is a well known crosscutting concern in JHotDraw, it is again the Observer concern.

3.2. 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. It is thus classified as a utility method by Marin et al.

The second largest candidate is somewhat interesting: It consists of 12 invocations of different methods after a call to "*...AbstractCommand.execute*" (i.e. it is always called before one of the 12 other methods is invoked). 11 of the invocations are calls to the "*createUndoActivity*" method of 11 different classes. The other is an invocation of "*...ZoomDrawingView.zoomView*", which could be interpreted as an *anomaly*—maybe this deviating behavior is related to a bug in the program. However, the other 11 invocations are of classes representing operations that change the figure and *zoomView* (probably) does not change it, thus this is not an anomaly.

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, these methods can be regarded as utility methods.

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). It is part of the Observer concern.

size	candidates	size	candidates
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 3. Filtered Inside-First Execution Relations

size	candidates
2	11
3	3
4	2
5	1
7	1
12	1

62 relations ($R^{\rightarrow}$) in 19 candidates

Table 4. Filtered Outside-Before Execution Relations

3.3. A simple filter

We have seen in the last section that most of the discovered cross cutting concerns are not to be refactored, because they are perfectly valid in their characteristics: Most of them are based on heavy use of delegation. While delegation can be often be regarded as crosscutting, we want to identify cross cutting 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 is to use the signatures of the invoked methods. It is based on the assumption that any method that returns a value has been delegated a part of the calling method's task and the results of the delegated tasks are immediately needed by the delegation method.

Only void methods are not directly needed where they are invoked. Of course, this is over-simplifying because of reference parameters. The return value of non-void methods is used by the calling method, thus, non-void methods are usually delegations from the calling method and not refactorable into aspects. Marin [13] has used non-void methods as parts of crosscutting concerns, for example the Undo concern. However, this refactoring also extracted the enclosing statements in a complex refactoring. The goal of our approach is to identify simple possibilities for refactoring.

The implemented filter extracts only those uniform and crosscutting execution relations that call a void method. The results of this filter are shown in Table 3 and 4.

Figure 2 shows the extracted cross cutting inside-first-execution relations for candidates with at least seven relations. A closer look at these relations reveal that most of them have the characteristics of cross cutting concerns, especially the larger ones. We will now discuss them in detail:

...AbstractFigure.willChange This method has been discussed in the previous sections and is part of the Observer concern.

...AbstractCommand.execute This method has already been discussed in the previous section. This method is part of two known concerns in JHotDraw: the Command and the Contract enforcement concern [14].

...UndoableAdapter.setUndoable This is another method that belongs to the known Undo concern. However, as this is a setter method that just sets a field of the UndoableAdapter class, it is ignored in the fan-in approach of Marin et al. [14].

...AbstractTool.mouseDown has been already identified by Marin et al. as part of a consistent behavior concern.

...Rectangle.add This method belongs to the AWT standard library. It has not been observed by Marin et al. This method reveals a consistent behavior concern: At the 10 identified call sites a new rectangle is created from two points that have been provided as arguments to the current method. It is clearly a crosscutting concern that can be refactored (and probably should).

. . . ObjectInputStream.defaultReadObject This method is called within eight other methods, all specific versions of *readObject*. It is used to deserialize an object, which is read from an input stream. Thus, this can be classified as a consistent behavior concern and can be refactored into an aspect. Because this method is part of the standard library, this concern has not been described by Marin et al.

. . . Rectangle.translate This method moves a rectangle object by a given vector. As can be seen, it is used seven times in basicMoveBy methods (and there it is most often the only method called). It is clearly a delegation, but can also be seen as crosscutting behavior in the form of a consistent behavior concern. However, the translate method is used at least in 20 of more than 30 basicMoveBy methods. In many methods some other operations must be executed before the rectangle can be moved. Some others just execute the basicMoveBy method of the super class. Thus, the consistent be-

7 **...figures.AttributeFigure.read** $\in_\top$
...figures.EllipseFigure.read
...figures.RoundRectangleFigure.read
...figures.TextFigure.read
...figures.ImageFigure.read
...contrib.TextAreaFigure.read
...contrib.PolygonFigure.read
...figures.RectangleFigure.read
7 **...figures.AttributeFigure.write** $\in_\top$
...figures.ImageFigure.write
...contrib.TextAreaFigure.write
...figures.TextFigure.write
...figures.EllipseFigure.write
...contrib.PolygonFigure.write
...figures.RectangleFigure.write
...figures.RoundRectangleFigure.write
8 **java.awt.Rectangle.translate** $\in_\top$
...contrib.ComponentFigure.basicMoveBy
...figures.EllipseFigure.basicMoveBy
...contrib.TextAreaFigure.basicMoveBy
...figures.NullFigure.basicMoveBy
...figures.RectangleFigure.basicMoveBy
...samples.pert.PertFigure.basicMoveBy
...figures.ImageFigure.basicMoveBy
...figures.RoundRectangleFigure.basicMoveBy
8 **java.io.ObjectInputStream.defaultReadObject** $\in_\top$
...figures.ImageFigure.readObject
...standard.StandardDrawingView.readObject
...contrib.TextAreaFigure.readObject
...figures.LineConnection.readObject
...standard.StandardDrawing.readObject
...figures.TextFigure.readObject
...standard.DecoratorFigure.readObject
...standard.CompositeFigure.readObject
10 **java.awt.Rectangle.add** $\in_\top$
...figures.NullFigure.basicDisplayBox
...contrib.SimpleLayouter.calculateLayout
...samples.pert.PertFigure.basicDisplayBox
...contrib.zoom.AreaTracker.rubberBand
...standard.SelectAreaTracker.rubberBand
...figures.RectangleFigure.basicDisplayBox
...figures.EllipseFigure.basicDisplayBox
...figures.ImageFigure.basicDisplayBox
...figures.RoundRectangleFigure.basicDisplayBox
...contrib.ComponentFigure.basicDisplayBox
11 **...standard.AbstractTool.mouseDown** $\in_\top$
...standard.ActionTool.mouseDown
...figures.ScribbleTool.mouseDown
...standard.DragTracker.mouseDown
...standard.HandleTracker.mouseDown
...samples.javadraw.URLTool.mouseDown
...contrib.zoom.ZoomTool.mouseDown
...standard.CreationTool.mouseDown
...standard.ConnectionTool.mouseDown
...contrib.dnd.DragNDropTool.mouseDown
...contrib.PolygonTool.mouseDown
...standard.SelectionTool.mouseDown
13 **...util.UndoableAdapter.setUndoable** $\in_\top$
...standard.SendToBackCommand.UndoActivity.(init)
...figures.RadiusHandle.UndoActivity.(init)
...figures.BorderTool.UndoActivity.(init)
...figures.PolyLineHandle.UndoActivity.(init)
...standard.ResizeHandle.UndoActivity.(init)
...standard.ChangeConnectionHandle.UndoActivity.(init)
...figures.GroupCommand.UndoActivity.(init)
...figures.InsertImageCommand.UndoActivity.(init)
...standard.PasteCommand.UndoActivity.(init)
...figures.UngroupCommand.UndoActivity.(init)
...contrib.TriangleRotationHandle.UndoActivity.(init)
...contrib.PolygonScaleHandle.UndoActivity.(init)
...standard.SelectAllCommand.UndoActivity.(init)
17 **...standard.AbstractCommand.execute** $\in_\top$
...util.UndoCommand.execute
...standard.BringToFrontCommand.execute
...standard.CutCommand.execute
...figures.InsertImageCommand.execute
...standard.ToggleGridCommand.execute
...standard.AlignCommand.execute
...contrib.zoom.ZoomCommand.execute
...standard.CopyCommand.execute
...standard.DuplicateCommand.execute
...standard.DeleteCommand.execute
...standard.SelectAllCommand.execute
...util.RedoCommand.execute
...standard.ChangeAttributeCommand.execute
...figures.UngroupCommand.execute
...figures.GroupCommand.execute
...standard.PasteCommand.execute
...standard.SendToBackCommand.execute
20 **...standard.AbstractFigure.willChange** $\in_\top$
...contrib.TextAreaFigure.setFont
...contrib.GraphicalCompositeFigure.update
...contrib.TriangleFigure.rotate
...figures.TextFigure.moveBy
...contrib.PolygonFigure.scaleRotate
...contrib.PolygonFigure.setPointAt
...figures.LineConnection.startPoint
...contrib.PolygonFigure.removePointAt
...figures.ElbowConnection.updatePoints
...figures.TextFigure.setFont
...standard.AbstractFigure.moveBy
...figures.LineConnection.endPoint
...contrib.PolygonFigure.smoothPoints
...figures.PolyLineFigure.removePointAt
...figures.PolyLineFigure.setPointAt
...standard.AbstractFigure.displayBox
...contrib.html.HTMLTextAreaFigure.figureChanged
...contrib.TextAreaFigure.moveBy
...figures.RoundRectangleFigure.setArc
...contrib.PolygonFigure.insertPointAt

Figure 2. Extracted Inside-First-Execution Relations

havior concern is better not refactored into an aspect (otherwise, the basicMoveBy methods would be very heterogeneous). In contrast, this specific concern for the movement of figures has also been identified by Ceccato et al. [6], where it has been classified as an "aspectizable", i.e. can be refactored into an aspect.

...AttributeFigure.write and ...AttributeFigure.read These two methods are responsible to read figures from an input stream and write them to an output stream resp. Because figures can be composed, the composite figures delegate reading and writing to the embedded figures—it is an instance of the composite design pattern. It is more or less another instance of the consistent behavior concern, however, not to be regarded as an aspect to be refactored.

4. Related Work

Aspect mining has been identified as useful technique to understand crosscutting behavior in non-aspect-oriented programs and as an aid to help in refactoring non-aspect-oriented programs to aspect-oriented programs. Most of the early approaches were not automatic and one has to specify a pattern that can be searched for in the source code [8, 23].

The most similar approach to ours is the approach of Marin et al. [14], who use fan-in analysis to identify crosscutting concerns. Fan-in analysis basically counts for each method the number of call sites in the source code that call the method. This approach is very similar to our inside-first- and inside-last-execution relations, however, our approach is more specific as it only identifies candidates that are easily refactorable by advice.

Gybels and Kellens [9, 10] use heuristics to mine for crosscutting concerns. The "unique methods" heuristic is defined as "A unique method is a method without a return value which implements a message implemented by no other method" and can be compared to our non-void methods filter. Gybels and Kellens also search for (unique) methods that are called from many places.

Tourwe and Mens [21] 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. Tonella and Ceccato [20] also use concept analysis for aspect mining, but they apply it on traces generated by dynamic analysis.

Some other approaches rely on clone detection techniques to detect scattered code in the form of crosscutting concerns: Bruntink [4, 3, 5] 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 by Shepherd et al. [15] uses a program dependence graph based clone detection technique for aspect mining. After an initial phase that detects clones, a second step filters the candidates, and a third phase coalesces the remaining candidates.

Remaining approaches use Natural Language Processing to analyze the identifiers used in source code [18] and clustering of related methods [17].

Ceccato et al. [6] have done a comparison of three aspect mining approaches: fan-in analysis [14], identifier analysis [21], and dynamic analysis [20]. The differences in the approaches and their results are presented and examined.

Timna by Shepherd et al. [16] is a framework for the combination of aspect mining techniques with the goal to increase precision and recall in comparison to approaches that use a single technique.

Binkley et at. [1] presents a semi-automated approach to refactor identified crosscutting concerns in object-oriented programs into aspects. Marin [13] has described how the Undo concern in JHotDraw has been refactored into an aspect manually.

Engler et al. [7] use statistical analysis to infer consistent and deviant behavior based on paired calls that follow one another. The paired calls are very similar to our outside-before and -after execution relations and we could use statistical analysis to find more crosscutting anomalies as presented in the Section 3.2.

5. Discussion, Conclusions, and Future Work

This evaluation of the static aspect mining tool has shown that most of the unfiltered and filtered identified crosscutting candidates are not concerns refactorable into aspects. This is not much different from results in our previous dynamic aspect mining approach [2]. 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.

Because of the small number of analyzed candidates in a single test program, the results cannot be generalized. However, based on the previous results from the dynamic approach and the comparison to other mining approaches for the analyzed program, our hypothesis is that the results will not change and are general. This would mean that aspect mining will have a hard time to identify candidates that are really refactorable into aspects. This hypothesis is in line with other results from similar studies, for example, Marin et al. used a large set of utility methods that are filtered out [14]. Moreover, the ongoing refactoring of JHotDraw into a system that makes good use of aspect oriented programing shows that a refactoring is usually a complex task

[13]. However, in contrast to other authors, we believe that most detected crosscutting concern in any aspect mining approach will reveal delegations that should not be refactored into aspects. Delegation can be regarded as a simple form of crosscutting, however, only superimposing delegations that are loosely coupled to the surrounding code can be refactored into aspects.

Therefore, future work will continue in two directions:

1. A large-scale analysis of discovered candidates for a larger set of programs with static and dynamic analysis. This includes the analysis of two other systems that have been used in other aspect mining approaches (Tomcat and PetStore).
2. Development of a filter which extracts the refactorable candidates from the discovered candidates. The presented simple filter already generates good results, however, exception handling can disturb the applicability (like in the inside-last-execution relations) and should be filtered.

References

[1] 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'05)*, pages 27–36, 2005.

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

[3] M. Bruntink. Aspect mining using clone class metrics. In *Workshop on Aspect Reverse Engineering*, 2004.

[4] 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.

[5] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwe. On the use of clone detection for identifying crosscutting concern code. *IEEE Trans. Softw. Eng.*, 31(10):804–818, Oct. 2005.

[6] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwe. A qualitative comparison of three aspect mining techniques. In *13th International Workshop on Program Comprehension (IWPC)*, 2005.

[7] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.*, 35(5):57–72, 2001.

[8] 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.

[9] K. Gybels and A. Kellens. An experiment in using inductive logic programming to uncover pointcuts. In *First European Interactive Workshop on Aspects in Software*, 2004.

[10] K. Gybels and A. Kellens. Experiences with identifying aspects in smalltalk using 'unique methods'. In *Workshop on Linking Aspect Technology and Evolution (LATE)*, 2005.

[11] 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.

[12] J. Krinke and S. Breu. Control-flow-graph-based aspect mining. In *Workshop on Aspect Reverse Engineering*, 2004.

[13] M. Marin. Refactoring jhotdraw's undo concern to aspectj. In *Workshop on Aspect Reverse Engineering (WARE)*, 2004.

[14] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE2004)*, 2004.

[15] D. Shepherd, E. Gibson, and L. Pollock. Design and evaluation of an automated aspect mining tool. In *International Conference on Software Engineering and Practice*, 2004.

[16] D. Shepherd, J. Palm, L. Pollock, and M. Chu-Carroll. Timna: A framework for combining aspect mining analyses. In *International Conference on Automated Software Engineering*, 2005.

[17] D. Shepherd and L. Pollock. Interfaces, aspects, and views. In *Workshop on Linking Aspect Technology and Evolution (LATE)*, 2005.

[18] D. Shepherd, T. Tourwe, and L. Pollock. Using language clues to discover crosscutting concerns. In *First International Workshop on the Modeling and Analysis of Concerns in Software (MACS)*, 2005.

[19] 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.

[20] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In *11th IEEE Working Conference on Reverse Engineering (WCRE 2004)*, 2004.

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

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

[23] 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.