Mining Aspects from CVS Transactions using Concept Analysis

Silvia Breu
University of Cambridge, UK
silvia@ieee.org

Thomas Zimmermann Christian Lindig
Saarland University, Germany
{zimmerth,lindig}@cs.uni-sb.de

1 Motivation

Every large program contains a small fraction of functionality that resists proper encapsulation. Code for debugging, logging, or locking is hard to conceal using object-oriented mechanisms alone. As a result, this code ends up scattered across many classes, which makes it a maintenance problem. At the same time, this code is largely orthogonal to surrounding (or *mainline*) code as it rarely impacts control or data flow. This observation gave rise to aspect-oriented programming (AOP) as a solution: Functionality is encapsulated in so-called aspects that are woven into mainline code during compilation.

For existing projects to benefit from AOP, the cross-cutting concerns must be identified first; this task is called *aspect mining*. In this paper we address it based on the hypothesis that a typical cross-cutting concerns is added to a project within a short amount of development time: We mine CVS archives for *sets of methods* that got *added together* in various *unrelated locations*. To compute these efficiently, we apply formal concept analysis [3]—an algebraic theory. In this paper, we describe the basic idea and report on results from an initial evaluation of our technique.

2 Mining Cross-Cutting Concerns

In our approach we first collect the data that represents the history of a project, namely all transactions to the CVS archive of the project. In a second step, we use formal concept analysis to mine aspect candidates from each transaction.

2.1 Version Archives and Transactions

The history of a project is characterized by a sequence of CVS *transaction*. Each transaction represents the changes, i.e. addition and deletions, between the previous and the current version. Motivated by our previous dynamic aspect mining approaches that analysed program traces [1, 2], we are only interested in changes that insert (or delete) calls to methods. As we are interested in the introduction of cross-cutting concerns (due to our hypothesis that aspects emerge over time) we omit deletions of method calls and concentrate exclusively on additions of method calls. A method call is characterized by two components: a *location* $l \in \mathcal{L}$ where the call originates (in the body of a method) and the *method* $m \in \mathcal{M}$ being called.

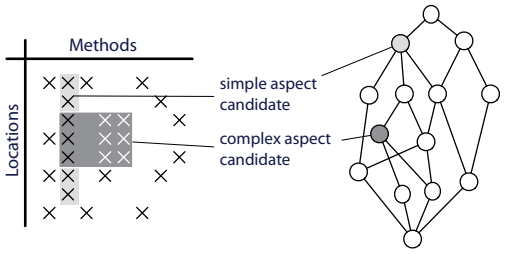

Figure 1: Maximal blocks represent aspect candidates in a transaction (left). Here, 14 candidates form a lattice of super and sub aspects (right).

Certain additions of method calls are suspect: when a small set $M \subseteq \mathcal{M}$ of (related) methods calls is added in many (unrelated) locations $L \subseteq \mathcal{L}$, this addition is likely to introduce a cross-cutting concern. We therefore represent an aspect candidate as a pair (L, M) of locations and methods. Our goal is to identify the most promising such pairs for each transaction.

2.2 Formal Concept Analysis

A single transaction of method call additions can be represented as a table with method locations as rows, and method calls as columns: Has a method call m been added in a location l, the intersection of m and l is marked by a cross in the table (Fig. 1, left).

As an aspect candidate is a pair (L, M) of locations and methods, it is represented in the table by a rectangle of crosses which is a *maximal block*, e.g., see the grey-shaded rectangles in Fig. 1. To make these blocks visible it is necessary to re-order rows and columns.

It is easy to identify the addition of calls to a single method—these are represented by $n \times 1$ blocks. No obvious solution exist to identify all locations where two methods like `lock` and `unlock` were added when these names are not known beforehand. To identify such a $n \times 2$ block, both rows and columns need to be re-ordered.

Identifying *all* maximal blocks in a cross table (or transaction) $T \subseteq \mathcal{L} \times \mathcal{M}$ is provided by the algebraic theory of formal concepts. A maximal block (or aspect candidate) is a pair (L, M) where the following holds:

$$\begin{aligned} L &= \{l \in \mathcal{L} \mid (m, l) \text{ for all } m \in M\} \\ M &= \{m \in \mathcal{M} \mid (m, l) \text{ for all } l \in L\} \end{aligned}$$

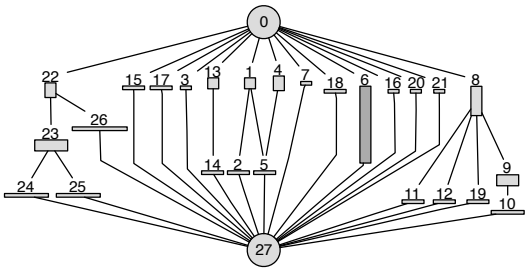

Figure 2: Lattice of aspect candidates from a commit to Eclipse CVS on 2004-03-01 by developer `ptff`.

Each block (L, M) is maximal in the following sense: We cannot add another method m to M without shrinking L to ensure that *all* locations in L call m. Likewise, we cannot add another location l to L without shrinking M. The definition allows for blocks of any size. However, we consider only a block that cross-cuts at least 7 locations an aspect candidate.

The blocks of a transaction form a lattice given the following partial order: $(L, M) \leq (L', M')$ iff $L \subseteq L'$. A sub block cross-cuts fewer locations than its super block but calls more methods (c.f. Fig. 1). The maximal blocks of a transaction and their lattice may be computed efficiently [4] and we use this to compute all aspect candidates (L, M).

3 First Application

In an initial evaluation we applied our analysis to the Eclipse as well as to the ArgoUML CVS repository. First results are promising and support our hypothesis that specific cross-cutting concerns may be identified by analyzing CVS transations. The following examples gives a first impression.

Figure 2 shows the lattice of all blocks of a Eclipse CVS commit transaction on 2004-03-01. In the lattice two blocks are connected if they are in a direct super/sub-block relation. Nodes are given the shape of the corresponding block which gives prominence to large aspect candidates: For example, candidate 6 contains 14 location where calls to `unsupportedIn2()` were added. This method throws an exception if the operation called is not supported at API level 2.0.

The second example we want to present here is from ArgoUML. The transaction with the log message *"Made the methods look a little more alike. Collected the numerous IllegalArgument calls in methods. [...]"* inserted many cross-cutting calls to `illegalArgument` or one of its variants. These calls are always last in the method body:

```
public String getValueOfTag(Object handle) {
  if (handle instanceof MTaggedValue) {
    return ((MTaggedValue) handle).getValue();
  }
  return illegalArgumentString(handle);
}
```

In this case the method `illegalArgumentString` throws an `IllegalArgumentException` and returns a `null` object. Most of the 262 calls to `illegalArgument` methods could have been realized as aspects.

4 Conclusions

When calls to a small set of functions are added to many locations, this is likely to represent a cross-cutting concern, especially when such an addition happens in a single CVS transaction. We leverage this observation to mine aspect candidates from CVS repositories. Because transactions are typically small and we mine one transaction at a time, our mining scales well: we are the first to report aspects for Eclipse, which comprises over 1.6 million lines of code and about 13 000 classes—which is our first contribution.

The identification of aspects is based on co-addition: the same set of calls is added in many locations. We compute all co-additions of a transaction efficiently using concept analysis and identify the most likely aspects. Concept analysis provides a conceptual and algorithmic framework to identify co-additions and thus aspect candidates—which is our second contribution.

References

[1] S. Breu. Extending Dynamic Aspect Mining with Static Information. In *Proceedings of 5th International Workshop on Source Code Analysis and Manipulation (SCAM)*, pp. 57–65. IEEE Computer Society, September/October 2005.

[2] S. Breu and J. Krinke. Aspect Mining Using Event Traces. In *Proceedings of 19th International Conference on Automated Software Engineering (ASE)*, pp. 310–315. IEEE Press, September 2004.

[3] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations.* Springer, Berlin – Heidelberg – New York, 1999.

[4] C. Lindig. Fast concept analysis. In G. Stumme, editor, *Working with Conceptual Structures – Contributions to ICCS 2000*, pp. 152–161, Germany, 2000. Shaker Verlag.

[5] B. Livshits and T. Zimmermann. DynaMine: finding common error patterns by mining software revision histories. In *Proceedings of European Software Engineering Conference/ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 296–305, New York, NY, USA, 2005. ACM Press.

[6] C. C. Williams and J. K. Hollingsworth. Recovering system specific rules from software repositories. In *Proceedings of the Intl. Workshop on Mining Software Repositories*, pp. 7–11, May 2005.