Aspect Mining for Large Systems

— Poster —

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

Thomas Zimmermann
Saarland University
Dept. of Computer Science
Saarbrücken, Germany
tz@acm.org

Christian Lindig
Saarland University
Dept. of Computer Science
Saarbrücken, Germany
lindig@cs.uni-sb.de

Abstract

As software evolves, new functionality sometimes no longer aligns with the original design, ending up scattered across a program. We find such cross-cutting concerns by applying formal concept analysis to the program's history: method calls added across many locations are likely to be cross-cutting. Our approach scales up to Eclipse.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—Restructuring, reverse engineering, and reengineering

General Terms Algorithms, Measurement, Documentation, Performance, Design, Experimentation,

Keywords Analyzing Version Archives, Aspect-Oriented Programming, Eclipse, Formal Concept Analysis, Java, Aspect Mining

1. Introduction

As object-oriented programs grow, new functionality sometimes no longer aligns with the initially chosen design and modularization. Every large program contains a small fraction of functionality that resists clean encapsulation. Code for debugging, logging, or locking is hard to keep hidden 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: cross-cutting functionality is factored out into so-called aspects and these are woven back into mainline code during compilation.

However, for existing software systems to benefit from AOP, cross-cutting concerns must be identified first. Only then a system can be re-factored into an aspect-oriented design. This identification task is called *aspect mining*. Previous approaches to aspect mining applied static or dynamic program analysis techniques to a single version of a program. As a result, they often have difficulties to scale to large systems: Dynamic analysis depends on many test cases and static analysis is hard to implement as an incremental analysis. We solve this problem by treating a system's version history as mining ground for aspects. Our new approach analyzes changes from one version to the next and thus is independent from the total size of a system. And since it is static, it does not rely on test cases but guarantees complete coverage. As a result, we were able to mine aspects from Eclipse, a project with over 1.3 million lines of code for which we analyzed over 40 000 CVS transactions.

Our mining builds on the hypothesis that cross-cutting functionality (or aspects) emerge over time. We analyze where method calls are added from one version of a system to the next. A method call is likely to introduce an aspect if the same method call is added in many method bodies (which we call locations). An aspect is even more likely to be present when the same two, three, or more method calls are added in many locations: calls to `lock` and `unlock` are a typical example.

2. Preprocessing

Our approach can be applied to any version control system. However, we based our implementation on CVS since most open source projects currently use it. First, we reconstruct CVS commits with a *sliding time window* approach [4]. A reconstructed commit consists of a set R of revisions where each revision $r \in R$ is the result of a single check-in.

Additionally, we compute method calls that have been inserted within a commit operation R. A commit R is a set of changed locations—in our case locations are method bodies but could be classes or packages as well. For every location l that was changed in R we compute the set $M(l)$ of added method calls by comparing the abstract syntax tree of l before and after commit R. As a result we obtain a set $T(R) = \{(l, m) \mid l \in R, m \in M(l)\}$ of *new calls* from location l to method m. We call a set $T(R)$ of new calls a *transaction*; transactions serve as main input for our aspect mining. Here is an example from the Eclipse project:

$$\left\{\begin{array}{l}(\texttt{DefaultByteCodeVisitor.\_aaload(int)}, \texttt{dumpPcNumber(1)}),\\ (\texttt{DefaultByteCodeVisitor.\_aastore(int)}, \texttt{dumpPcNumber(1)}),\\ (\texttt{DefaultByteCodeVisitor.\_aload(int, int)}, \texttt{dumpPcNumber(1)})\end{array}\right\}$$

Into three locations `_aaload`, `_aastore`, and `_aload` a call to method `dumpPcNumber(1)` was inserted. In order to reduce computational cost, we analyze only the differences between single revisions but not between the resulting programs before and after a revision. Therefore we cannot resolve signatures for called methods. Instead we use their names (e.g., `dumpPcNumber`) and number of arguments (e.g., 1).

3. Mining Transactions

For our analysis, the history of a program is a sequence of transactions. Each transaction is a set of added method calls (l, m) from

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
ACM 1-59593-491-X/06/0010.

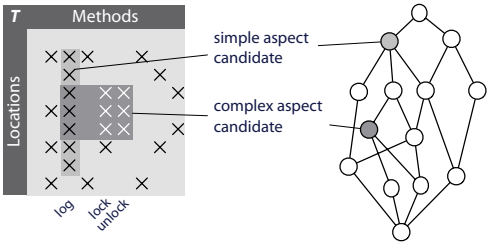

Figure 1. A transaction $T \subseteq \mathcal{L} \times \mathcal{M}$ is a relation between locations $\mathcal{L}$ and methods $\mathcal{M}$. The maximal (rectangular) blocks of T are aspect candidates, which form a hierarchy.

location l to method m[1]. A transaction T is formally a relation and can be depicted as a cross table between locations L and methods M—cf. Figure 1 .

Simple Aspects When a transaction inserts calls to a logging method `log` in 10 locations these calls show up in the cross table as a *block* of size 1×10 (given an appropriate order of locations). We consider adding a call to be an aspect candidate when it cross-cuts at least 8 locations. At each location where a call to `log` was added, calls to other methods may have been added as well. Still, aspects where a call to a single method (like `log`) is added are simple to detect in a transaction by sorting calls (l, m) by the called method m. We call these *simple aspect candidates*. Obviously a candidate is more likely to be a genuine aspect when the number of locations it cross-cuts is high.

Complex Aspects Some aspects come as pairs of function calls: a call to `lock` for locking a resource is typically followed by a call to `unlock`. Given an appropriate order of rows *and* columns, the addition of calls to `lock` and `unlock` in 10 locations also shows up as a (2×10) block in the cross table. We call the addition of calls to two or more methods a *complex aspect* candidate. Again, we consider such a block only a candidate if it cross-cuts at least 8 locations. Unlike simple aspect candidates, it is not obvious how to detect such complex aspect candidates in a transaction efficiently.

4. Formal Concept Analysis

The problem of identifying all blocks is the subject of formal concept analysis, an algebraic theory for binary relations [2], which also provides efficient algorithms [3]. A maximal block in a transaction $T \subseteq \mathcal{L} \times \mathcal{M}$ is a pair (L, M) of locations and methods where the following holds:

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

Formal concept analysis considers all blocks in a relation, not just those exceeding certain limits. The definition of blocks in particular allows for blocks with one empty component and subsumes simple and complex aspect candidates. We therefore compute all blocks and filter them later for aspect candidates.

Interestingly, blocks and therefore aspects form a lattice, defined by the partial order $(L, M) \leq (L', M') \Leftrightarrow L \leq L'$. However, typically the aspect candidates of a transaction are incomparable. Figure 2 shows the lattice of blocks for such a transaction from the Eclipse project.

[1] We ignore changes and deletions of calls as we are only interested in aspects emerging over time.

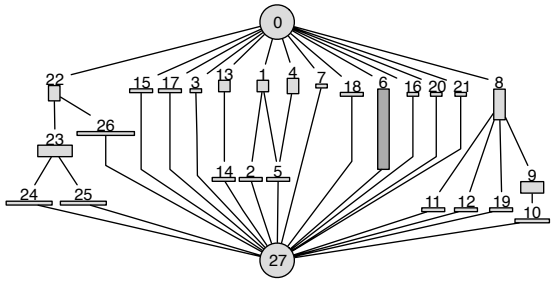

Figure 2. Hierarchy of blocks from an Eclipse transaction. Block 6 is an aspect candidate, cross-cutting 14 locations.

Aspect Candidates in Eclipse 3.2M3				
methods	1	2	3	≥ 4
candidates	1878	363	88	24

Table 1. Aspect candidates mined from 43 270 CVS transactions for Eclipse 3.2M3. There are 88 candidates that added exactly 3 method calls.

5. Experience and Results

Because a cross table of size $n \times n$ may have up to 2^n blocks, concept analysis is potentially expensive. This has not been a problem so far: for 43 270 transactions in the Eclipse CVS repository, the average transaction adds 5.4 calls in 3.8 locations and has 3.7 blocks. However, the largest transaction had 1235 blocks. On average, computing all blocks for a transaction took less than 1 second.

The 43 270 transactions of the Eclipse CVS archive constitute 159 448 blocks. From these we mined 2353 aspect candidates, with the distribution shown in Table 1. We found 1878 simple and $363 + 88 + 24 = 475$ complex candidates.

In [1] we had previously mined Eclipse for simple and complex aspect candidates, albeit with a less general approach. There we reported 31 unique complex candidates that cross cut at least 20 locations (out of which we found 6 to be true aspects and additional 3 to be partial aspects). With our new approach we found 64 unique aspect candidates, including all 31 aspect candidates reported in [1]. This confirms our two claims: formal concept analysis provides the right formal and algorithmic framework to mine aspects, and aspects can be mined efficiently from large projects by analyzing code additions over time.

References

[1] S. Breu and T. Zimmermann. Mining Aspects from Version History. *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*. IEEE Computer Society Press, 2006. Accepted for publication.

[2] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations.* Springer, Berlin, 1999.

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

[4] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proc. Intl. Workshop on Mining Software Repositories (MSR)*, Edinburgh, Scotland, May 2004.