Mining Aspects from Version History

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

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

Abstract

Aspect mining identifies cross-cutting concerns in a program to help migrating it to an aspect-oriented design. Such concerns may not exist from the beginning, but emerge over time. By analysing where developers add code to a program, our history-based aspect mining (HAM) identifies and ranks cross-cutting concerns.

We evaluated the effectiveness of our approach with the history of three open-source projects. HAM scales up to industrial-sized projects: for example, we were able to identify a locking concern that cross-cuts 1 284 methods in Eclipse. Additionally, the precision of HAM increases with project size and history; for Eclipse, it reaches 90% for the top-10 candidates.

1. Introduction

As object-oriented programs evolve over time, they may suffer from *"the tyranny of dominant decomposition"* [20]: They can be modularised in only one way at a time. Concerns that are added later may no longer align with that modularisation, and thus, end up scattered across many modules and tangled with one another. Aspect-oriented programming (AOP) remedies this by factoring out aspects and weaving them back in a separate processing step [10]. For existing projects to benefit from AOP, these cross-cutting concerns must be identified first. This task is called *aspect mining*.

We solve this problem by taking a historical perspective: we mine the history of a project and identify code changes that are likely to be cross-cutting concerns; we call them *aspect candidates*. Our analysis is based on the hypothesis that cross-cutting concerns evolve within a project over time. A code change is likely to introduce such a concern if the modification gets introduced at various locations within a single code change.

Our hypothesis is supported by the following example. On November 10, 2004, Silenio Quarti committed code changes "76595 (new lock)" to the Eclipse CVS repository. They fixed bug #76595 "Hang in gfk_pixbuf_new" that reported a deadlock[1] and required the implementation of a new locking mechanism for several platforms. The extent of the modification was enormous: He modified 2 573 methods and inserted in 1 284 methods a call to `lock`, as well as a call to `unlock`. As it turns out, AOP could have been used to add these.

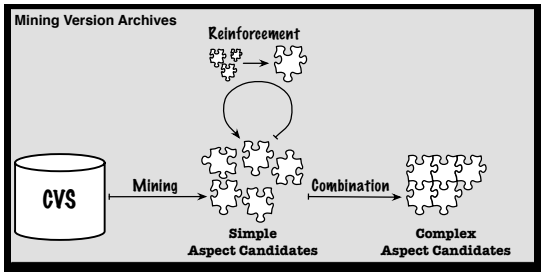


Figure 1. Mining workflow

Our approach searches such cross-cutting changes in the history of a program in order to identify aspect candidates. For Silenio Quarti's changes, we find two *simple aspect candidates* $(\{\texttt{lock}\}, L_1)$ and $(\{\texttt{unlock}\}, L_2)$ where L_1 and L_2 are sets that contain the 1 284 methods where `lock` and `unlock` have been inserted, respectively. It turns out that $L_1 = L_2$, hence, we combine the two simple aspect candidates into a *complex aspect candidate* $(\{\texttt{lock}, \texttt{unlock}\}, L_1)$.

Technically, we mine version archives for aspect candidates (see Figure 1). Our implementation HAM first identifies simple aspect candidates in transactions (Section 2). Next, we combine simple aspect candidates into complex ones that consider more than one method call (Section 4). We may get several aspect candidates for the same cross-cutting concern when it was added in several transactions. *Reinforcement* combines such candidates by exploiting *localities* between transactions (Section 3).

We evaluated HAM with three open-source Java projects: JHotDraw (57 360 LOC), Columba (103 094 LOC), and Eclipse (1 675 025 LOC). For each project we

[1] `https://bugs.eclipse.org/`

ranked candidates and validated the top-50 candidates manually. Our results are promising: the average precision is around 50% with the best values for Eclipse; for the top-10 candidates in Eclipse, HAM's precision is better than 90% (Section 6). Altogether, the contributions of this paper are as follows:

HAM adds a new dimension to aspect mining. Previous approaches considered only a particular version of a program. Our approach uses project history as additional input. This enables a new view on the evolution of cross-cutting concerns.

HAM scales and is platform independent. HAM is the first aspect mining approach that scales for industrial-sized projects like Eclipse. Furthermore, it recognises cross-cutting concerns across different platforms.

HAM comes with a high precision. We thoroughly evaluated 405 aspect candidates returned by HAM. The precision increases with the project size and history, for Eclipse up to 90% for the top-10 candidates.

2. Simple Aspect Candidates

Previous approaches to aspect mining considered only a single version of a program using static and dynamic program analysis techniques. Our approach introduces an additional dimension: the *history* of a project.

We model the history of a program as a sequence of transactions. A *transaction* collects all code changes between two versions, called *snapshots*, made by a programmer to complete a single development task. Technically a transaction is defined by the version archive we analyse, which is CVS in our case. However, our approach extends to arbitrary version archives.

Motivated by dynamic approaches for aspect mining that investigate execution traces of programs, we build our analysis on changes that insert or delete method calls. Typically, these changes have direct impact on execution traces. But since we are looking for the introduction of cross-cutting concerns, we concentrate solely on additions and omit deletions of method calls. This means that for our purpose a transaction consists of the set of method calls that were inserted by a developer.

Definition 1 (Transaction)
A transaction T *is a set of pairs* (m, l)*. Each pair* (m, l) *represents an insertion of a call to method* m *in the body of the method* l*.*

We name the method l into which a call is inserted *method location* and identify it by its full signature.

Algorithm 1 Simple aspect candidates

```
1: function CANDIDATES(T)
2:     C_result = ∅
3:     for all m ∈ calls(T) do
4:         L = {l | l ∈ locations(T), (m, l) ∈ T}
5:         C_result = C_result ∪ {(m, L)}
6:     end for
7:     return C_result
8: end function
9:
10: function SIMPLE_CANDIDATES(𝒯)
11:     return ⋃_{T∈𝒯} CANDIDATES(T)
12: end function
```

In contrast, to reduce the cost of preprocessing, we identify the called method m only by its name and number of arguments (see Section 5). We associate the following meta-data with a transaction T:

1. *developer*(T) is the name of developer who committed transaction T.
2. *timestamp*(T) is when a transaction T was committed.
3. *locations*$(T) = \{l \mid (m, l) \in T\}$ is the set of methods that were changed in transaction T.
4. *calls*$(T) = \{m \mid (m, l) \in T\}$ is the set of method calls that were added in transaction T.

Within the set $\mathcal{T}$ of transactions we are searching for *aspect candidates*. An aspect candidate represents a cross-cutting concern in the sense that it consists of one or more calls to methods M that are spread across several method locations L.

Definition 2 (Aspect Candidate)
An aspect candidate $c = (M, L)$ *consists of a non-empty set* M *of methods and a non-empty set* L *of locations where each location* $l \in L$ *calls each method* $m \in M$*. If* $|M| = 1$*, the aspect candidate* c *is called* simple*; if* $|M| > 1$*, it is called* complex*.*

Basically every method call m added in transaction T leads to a potential aspect candidate. Algorithm 1 reflects this idea in function SIMPLE_CANDIDATES($\mathcal{T}$) which returns for every transaction $T \in \mathcal{T}$ and every method call $m \in$ *calls*(T) one aspect candidate. The result would be huge for projects like Eclipse that have many method calls and a long history. Thus, we use filtering and ranking to find actual aspect candidates. In order to identify aspect candidates that actually cross-cut a considerable part of a program, we ignore all candidates $c = (M, L)$ where less than eight locations are cross-cut, i.e., $|L| < 8$. Thus, we get large, homogeneous cross-cutting concerns. We focus on them

as maintenance will benefit most from their modularisation in aspects. We chose the cut-off value of eight based on our previous experience [13]; for some projects lower cut-off values may be required. In addition to filtering, we use the following ranking techniques:

Rank by Size. Obviously, candidates that cross-cut many locations could be more interesting. Thus, we sort aspect candidates $c = (M, L)$ by their size $|L|$ (from large to small). However, we may get noise in form of method calls that are frequent in Java but are not cross-cutting, e.g., `iter()`, `hasNext()`, or `next()`.

Rank by Fragmentation. This ranking penalises common Java method calls when they appear in many transactions. If a cross-cutting concern is added to a system and not changed later on, it appears in only one transaction. To capture such aspects, we sort aspect candidates by the number of transactions in which we find a candidate (fewer is better). We term this count the *fragmentation* of an aspect candidate $c = (M, L)$:

$$\mathit{fragmentation}(c) = |\{T \in \mathcal{T} \mid M \subseteq \mathit{calls}(T)\}|$$

In case aspect candidates have the same fragmentation because they occur in the same number of transactions, we rank additionally by size $|L|$.

Rank by Compactness. Similar to the ranking by fragmentation, this ranking has the advantage that common Java method calls are ranked low. Cross-cutting concerns may be introduced in one transaction and extended to additional locations in later transactions. Since such concerns will be ranked low with the previous rankings, we use *compactness* as a third ranking technique (from high to low). The compactness of an aspect candidate $c = (M, L)$ is the ratio between the size $|L|$ and the total number of locations where calls to M occurred in the history:

$$\mathit{compactness}(c) = \frac{|L|}{|\{l \,|\, \exists T \in \mathcal{T}, \forall m \in M\colon (m, l) \in T\}|}$$

In the case that two or more aspect candidates have the same compactness, we rank additionally by size $|L|$.

3. Locality and Reinforcement

In our experiments, we observed that several cross-cutting concerns were introduced within one transaction and later inserted to other new locations in later transactions. We refer to this as "extending a cross-cutting concern to new locations later". This happens frequently when a developer recognises he must complete a task that he had left unfinished with his last commit. Although such concerns are recognised by our technique as multiple aspect candidates, these candidates may be ranked low and missed.

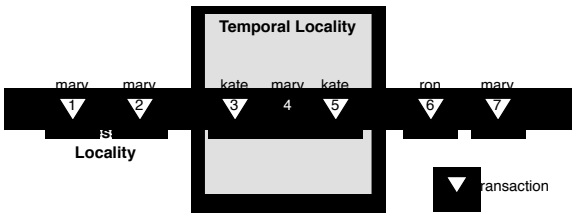


Figure 2. Possessional and temporal locality for transaction 4.

To strengthen aspect candidates that were inserted in several transactions, we use the concept of *locality*. Two transactions are locally related if they were created by the same developer or were committed around the same time. If there exists locality between transactions, we *reinforce* their aspect candidates mutually.

Temporal Locality refers to the fact that aspect candidates may appear in several transactions that are close in time. In Figure 2 there exists temporal locality between transaction 4 and transactions 3 and 5.

Possessional Locality refers to the fact that aspect candidates may have been created by one developer but committed in different transactions; thus they are *owned* by her. Girba et al. [5] define ownership by the last change to a line; in contrast, we look for the addition of method calls, which is more fine-grained. In Figure 2 there exists possessional locality between transaction 4 and transactions 1, 2, and 7, all of them were committed by Mary.

Definition 3 (Locality)
Let $T_1, T_1 \in \mathcal{T}$ be arbitrary transactions, $c = (M, L)$ be an aspect candidate, and t be a fixed time interval. We say T_1 and T_2 have

(a) temporal locality*, written as $T_1 \overset{time}{\rightsquigarrow} T_2$ iff*

$$|\mathit{timestamp}(T_1) - \mathit{timestamp}(T_2)| \leq t$$

(b) possessional locality*, written as $T_1 \overset{dev}{\rightsquigarrow} T_2$ iff*

$$\mathit{developer}(T_1) = \mathit{developer}(T_2)$$

Presume that we found two aspect candidates $c_1 = (M_1, L_1)$ and $c_2 = (M_2, L_2)$ in two different transactions where the called methods are the same, i.e., $M_1 = M_2$. If there exists locality of either form between these two transactions, we can combine both aspect candidates. As a result we get a new aspect candidate $c' = (M_1, L_1 \cup L_2)$. We call this process *reinforcement*.

Algorithm 2 Reinforcement algorithms

```
1: function REINFORCE(T, x ∈ {time, dev})
2:     C_reinf = ∅
3:     for all T ∈ T do
4:         T_loc = { T' | T' ∈ T, T' ~x~> T }
5:         C_loc = ⋃_{T'∈T_loc} CANDIDATES(T')
6:         for all c = (M, L) ∈ CANDIDATES(T) do
7:             L_reinf = {L' | c' = (M', L') ∈ C_loc, M' = M}
8:             C_reinf = C_reinf ∪ {(M, L_reinf)}
9:         end for
10:    end for
11:    return C_reinf
12: end function
13:
14: function TEMPORAL(T)
15:    return REINFORCE(T, time)
16: end function
17:
18: function POSSESSIONAL(T)
19:    return REINFORCE(T, dev)
20: end function
21:
22: function ALL(T)
23:    return TEMPORAL(T) ∪ POSSESSIONAL(T)
24: end function
```

Definition 4 (Reinforcement)
Let $c_1 = (M_1, L_1)$ and $c_2 = (M_2, L_2)$ be aspect candidates. If $M_1 = M_2$, the construction of a new aspect candidate $(M, L_1 \cup L_2)$ with $M = M_1 = M_2$ is called reinforcement.

We implemented three reinforcement algorithms, which are listed in Algorithm 2. The functions for temporal (TEMPORAL) and for possessional (POSSESSIONAL) reinforcement both call function REINFORCE which

1. takes a set $\mathcal{T}$ of transactions as input,
2. identifies for each transaction T other transactions $\mathcal{T}_{loc}$ that are related to T with respect to the given locality x,
3. computes for each of these transactions the simple aspect candidates, and
4. builds new combined, or *reinforced* candidates.

Additionally, we implemented an algorithm ALL that combines the results of temporal and possessional reinforcement. However, it does not use the localities at the same time as this could reinforce all transactions and would thereby lose the historic perspective of our approach, but applies them independently.

4. Complex Aspect Candidates

Many cross-cutting concerns consist of more than one method call. An example is the `lock`/`unlock` concern

Algorithm 3 Complex aspect candidates

```
1: function COMPLEX_CANDIDATES(C_simple)
2:     C_result = ∅
3:     for all (M, L) ∈ C_simple do
4:         M = {M' | (M', L') ∈ C_simple, L = L'}
5:         M_complex = ⋃_{M'∈M} M'
6:         C_result = C_result ∪ {(M_complex, L)}
7:     end for
8:     return C_result
9: end function
```

presented in Section 1. To locate such concerns we combine two aspect candidates $c_1 = (M_1, L_1)$ and $c_2 = (M_2, L_2)$ to a complex aspect candidate $c' = (M', L')$ with $M' = M_1 \cup M_2$ and $L' = L_1$, if c_1 and c_2 cross-cut exactly the same locations, i.e., $L_1 = L_2$. This condition is very selective, however, method calls inserted in the same locations are very likely to be related.

Algorithm 3 constructs complex aspect candidates. Function COMPLEX_CANDIDATES takes all simple aspect candidates as input and combines candidates with matching method locations into a new complex aspect candidate. Note that it also combines simple aspect candidates that were inserted in different transactions.

5. Data Collection

Our mining approach can be applied to any version control system; however, we based our implementation on CVS since most open-source projects use it. One of the major drawbacks of CVS is that commits are split into individual check-ins and have to be reconstructed. For this we use a *sliding time window* approach [26] with a 200 seconds window. A reconstructed commit consists of a set of revisions R where each revision $r \in R$ is the result of a single check-in.

Additionally, we need to compute method calls that have been inserted within a commit operation R. For this, we build abstract syntax trees (ASTs) for every revision $r \in R$ and its predecessor and compute the set of all calls C_1 in r and C_0 for the preprocessor by traversing the ASTs. Then $C_r = C_1 \setminus C_0$ is the set of inserted calls within r; the union of all C_r for $r \in R$ forms a *transaction* $T = \bigcup_{r \in R} C_r$ which serves as input for our aspect mining and are stored in a database.

Since we analyse only differences between single revisions, we cannot resolve types because only one file is investigated at a time. In particular, we miss the signature of called methods; to limit noise that is caused by this, we use the number of arguments in addition to method names to identify methods calls. This heuristic is frequently used when analysing single files [13, 25]. We would get full method signatures when building

snapshots of a system. However, as Williams and Hollingsworth [23] point out, such interactions with the build environment (compilers, make files) are extremely difficult to handle, require manual interaction, and result in high computational costs. In contrast, our preprocessing is cheap, as well as platform- and compiler-independent.

Renaming of a method is represented as deleting and introducing several method calls. We thus may incidentally consider renamed calls as aspect candidates. Recognising such changes is known as *origin analysis* [6] and will be implemented in a future version of HAM. It will eliminate some false positives and improve precision.

6. Evaluation

In the introduction we told an anecdote how we identified cross-cutting concerns in the history of Eclipse. Another example for a cross-cutting concern is the call to method `dumpPcNumber` which was inserted to 205 methods in the class `DefaultBytecodeVisitor`. This class implements a visitor for bytecode, in particular one method for each bytecode instruction; the following code shows the method for instruction `aload_0`.

```
/**
 * @see IBytecodeVisitor#_aload_0(int)
 */
public void _aload_0(int pc) {
    dumpPcNumber(pc);
    buffer.append(OpcodeStringValues
      .BYTECODE_NAMES[IOpcodeMnemonics.ALOAD_0]);
    writeNewLine();
}
```

The call to `dumpPcNumber` can obviously be realised as an aspect. However, in this case aspect-oriented programming can even generate all 205 methods (including comment) since the methods differ only in the name of the bytecode instruction.

6.1. Evaluation Setup

For a more thorough evaluation we chose three Java open-source projects and mined them for cross-cutting concerns. Refer to Table 1 for some statistics.

JHotDraw 6.0b1 is a GUI framework to build graphical drawing editors.[2] We chose it for its frequent use as aspect mining benchmark.

Columba 1.0 is an email client that comes with wizards and internationalisation support.[3] We chose it because of its well-documented project history.

Eclipse 3.2M3 is an integrated development environment that is based on a plug-in architecture.[4] We chose it because it is a huge project with many developers and a large history.

For each project, we collected the CVS data as described in Section 5, mined for simple aspect candidates as defined in Section 2, reinforced them using the localities established in Section 3, and also built complex aspect candidates as introduced in Section 4. We investigated the following questions:

1. *Simple Aspect Candidates.* How precise is our mining approach? That is, how many simple aspect candidates are real cross-cutting concerns?

2. *Reinforcement.* It leads to larger aspect candidates, but does it actually rank true simple aspect candidates high, thus, improving precision?

3. *Ranking.* Can we rank aspect candidates such that more cross-cutting concerns are ranked first?

4. *Complex Aspect Candidates.* How many complex aspect candidates can we find by the combination of simple ones?

To measure *precision*, we computed for each project, ranking, and reinforcement algorithm the top 50 simple aspect candidates. In order to eliminate multiple evaluation effort due to possible duplicates, we combined these rankings into one set per project. For Columba we got 134, for Eclipse 159, and for JHotDraw 102 *unique* simple aspect candidates. Next, we sorted these sets alphabetically by the name of the called method in order to prevent bias in the subsequent evaluation. We used this order to classify simple aspect candidates manually into *true* and *false* cross-cutting concerns. The *precision* is then defined as the ratio of the number of true cross-cutting concerns to the number of aspect candidates that were uncovered by HAM. Precision is basically the accuracy of our technique's results and in general a common measure for search performance.

We considered an aspect candidate (M, L) as a true cross-cutting concern if it referred to the same functionality and the methods M were called in a similar way, i.e., at the same position within a method and with the same parameters. An additional requirement for a true cross-cutting concern was that it can be implemented using AspectJ. However, we did not take into account whether aspect-orientation is the best way to realise the given functionality. In cases of doubt, we classified a candidate as a false cross-cutting concern.

[2] `http://www.jhotdraw.org/`
[3] `http://www.columbamail.org/drupal/`
[4] `http://www.eclipse.org/`

Table 1. Evaluation subjects

	Columba	Eclipse	JHotDraw
Presence			
Lines of code	103 094	1 675 025	57 360
Java files	1 633	12 935	974
Java methods	4 191	74 612	2 043
History			
Developer	19	137	9
Transactions	4 105	97 900	269
– that changed Java files	3 186	77 250	241
– that added method calls	1 820	43 270	132
Method calls added	24 623	430 848	7 517
First transaction	2001-04-08	2001-05-02	2000-10-12
Last transaction	2005-11-02	2005-11-23	2005-04-25

Table 2. Precision of HAM (in %) for simple aspect candidates

	Columba	Eclipse	JHotDraw
Size	52	52	36
Fragmentation	46	54	30
Compactness	42	52	28

Table 3. The effect of reinforcement on the precision of HAM (in % points)

	Columba	Eclipse	JHotDraw
Temporal locality			
Size	+ 2	– 4	± 0
Compactness	+ 2	– 2	+ 4
Possessional locality			
Size	– 8	–20	+ 2
Compactness	+12	+ 8	+ 2
All localities			
Size	– 8	–20	+ 2
Compactness	+10	+ 6	+ 2

It would also be interesting to measure *recall*: the ratio of correctly identified aspect candidates and all candidates. Recall measures how well a search algorithm finds what is is supposed to find. However, determining recall values requires the knowledge of all aspect candidates—which is impossible for real-world software. We therefore cannot report recall numbers.

6.2. Simple Aspect Candidates

To evaluate our notion of simple aspect candidates we checked whether the top-50 candidates per ranking and project were cross-cutting or not. The precision as the ratio of true cross-cutting functionality and all (50) aspect candidates are listed in Table 2 for each project (columns) and each ranking (rows).

We observe that precision increases with subject size: It is highest for Eclipse and lowest for JHotDraw, the smallest subject. The ranking has a minor impact and no ranking is generally superior; the deviation among the precision values is at most 10 percentage points. Nevertheless, the ranking by size, which simply ranks by the number of locations where a method was added, seems to work well across all projects. It reaches a precision between 36 and 52 percent. Roughly speaking, every second (for JHotDraw every third) mined aspect candidate is a real cross-cutting concern.

Unlike ranking by size, ranking by fragmentation and by compactness take transactions or the number of overall modified locations into account. We believe that the poor performance of these rankings for our smaller subjects JHotDraw and Columba is caused by the much smaller number (hundreds/few thousands versus tens of thousands) of transactions and added method calls available for mining (see Table 1). In other words, we expect these rankings to benefit from long project histories. These generally correspond to many transactions, as they are present in Eclipse.

We can identify cross-cutting concerns with a precision between 36% and 54%; the precision increases with project size and history.

6.3. Reinforcement

After mining simple aspect candidates we evaluated the effect of reinforcement on them. Reinforcement takes a simple aspect candidate (M, L) from a single transaction and looks at locally related transactions in order to arrive at a candidate (M, L') with an enlarged set $L' \supset L$ of locations. For the evaluation we reinforced the simple aspects of our subjects using temporal, possessional, and contextual locality, and also using all localities applied at once. As before, we checked the top-50 aspect candidates and computed the precision.

Table 3 lists the *change in precision* for each subject (columns), each locality (rows), and each ranking by size or compactness (sub-rows). Changes are relative to the precision before reinforcement (Table 2). Hence, these changes express the effect of reinforcement on the precision of our mining.[5]

Temporal locality produces slight improvements but seems to be unsatisfying with large projects. We presume that this is because we chose the same fixed time window of 2 days for all three subjects; we plan to investigate whether a window size proportional to a project's size would yield better results. The Eclipse project has far more developers as well as CVS transactions per day than JHotDraw and Columba. Thus,

[5]Note that for reinforcement we did not rank by fragmentation. This ranking punishes reinforced aspect candidates that are spread across many transactions.

we have too much noise that diminishes the positive impact of temporal locality for Eclipse.

Possessional locality shows the most significant improvement. Albeit ranking by size decreases precision up to 20 percentage points, possessional locality in combination with ranking by compactness improves precision up to 12 percentage points for all three subjects. In large projects, `get` and `set` methods are inserted in many locations and thus alleviate the positive effects of possessional locality for Eclipse when aspect candidates are ranked by size.

All localities considers the application of both localities. The effect on the precision is the same as with reinforcement based on possessional locality only: ranking by size annihilates the positive impact, ranking by compactness facilitates it. Thus, possessional locality is dominant and affects precision prominently.

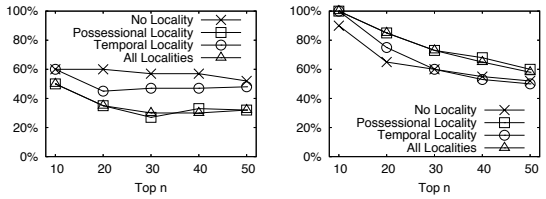
The good results for possessional locality suggest that aspects belong to a developer, and are mostly not distributed over many transactions. This is backed up by the notably improved precision of our approach after reinforcement based on possessional locality combined with ranking by compactness. Besides, all our results, without and with reinforcements, suggest that small projects have small histories and thus we achieve a significantly lower precision. In addition, precision can only be improved marginally with reinforcements. This seems consequential as reinforcements leverage a large amount of transactions and developers.

Possessional locality improves the precision for ranking by compactness; this indicates that cross-cutting concerns are owned by developers.

6.4. Precision Revisited

So far we have evaluated our mining by computing the precision of the top-50 aspect candidates in a ranking. However, it is unlikely that a developer is really interested in 50 aspect candidates. Instead, she will probably look only at ten or twenty candidates at most. We therefore have broken down the precision for the top ten, twenty, and so on candidates for each project. The results for all three subjects are similar. For the detailed discussion here, we have chosen Eclipse for two reasons—it is an industrial-sized project and the results are most meaningful; they are plotted in Figure 3.

The graph on the left shows the precision when *ranked by size* before and after applying different reinforcements. The precision stays mostly flat when moving from the top-50 to the top-10 candidates. However, the overall precision remains between 30 and 60 percent. Reinforcement seems to make matters only worse, as ranking by size before reinforcement performs best.



(a) Ranking by size (b) Ranking by compactness

Figure 3. Precision of HAM with respect to the length of ranking for subject Eclipse

In contrast, the graph on the right shows a dramatically different picture for the precision when *ranked by compactness*. The precision is highest for the top-10 candidates and decreases when additional candidates are taken into account; it is lowest for the top-50 candidates. However, the first ten candidates have a precision of at least 90%. This means, nine out of ten are true cross-cutting concerns. Thus, ranking by compactness is very valuable for developers.

In summary, size is not the most prominent attribute of cross-cutting concerns, but compactness is. This is also supported by the obversation that temporal and possessional locality enhance ranking by compactness.

Ranking by compactness pushes true cross-cutting concerns to the top such that we reach a precision of 90% for the top-10 candidates in Eclipse.

6.5. Complex Aspect Candidates

For our evaluation subjects, we combined simple aspect candidates into a complex candidate if they cross-cut exactly the same locations. This condition was very selective: for Columba we got 21, for Eclipse 178, and for JHotDraw 11 complex aspect candidates. Note that all candidates cross-cut at least 8 locations. Below, we discuss the results from Eclipse in more detail.

Table 4 shows the top 20 complex aspect candidates ranked by size for the Eclipse project. Each row represents one complex aspect candidate (M, L). The second column contains the methods M called by an aspect candidate, where the number in brackets denotes the number of arguments for each method. The third column gives the number $|M|$ of methods and the fourth column shows the number $|L|$ of method locations where calls to M were inserted. In the first column we provide the result of our manual inspection of this aspect candidate: ✓ for an actual cross-cutting concern and ✗ for a false positive.

HAM indeed finds cross-cutting concerns consisting of several method calls. In addition, they are ranked on top of the list. However, the performance of our

Table 4. Complex aspect candidates (M,L) found for Eclipse

M	$\|M\|$	$\|L\|$
✓ {lock(0), unlock(0)}	2	1284
✓ {postReplaceChild(3), preReplaceChild(3)}	2	104
✓ {postLazyInit(2), preLazyInit(0)}	2	78
✗ {blockSignal(2), unblockSignal(2)}	2	63
✓ {getLength(0), getStartPosition(0)}	2	62
✓ {hasChildrenChanges(1), visitChildrenNeeded(1)}	2	62
✗ {modificationCount(0), setModificationCount(1)}	2	60
✗ {noMoreAvailableSpaceInConstantPool(1), referenceType(0)}	2	57
✗ {g_signal_handlers_block_matched(7), g_signal_handlers_unblock_matched(7)}	2	54
✗ {getLocalVariableName(1), getLocalVariableName(2)}	2	51
✗ {isExisting(1), preserve(1)}	2	48
✗ {isDisposed(0), isTrue(1)}	2	37
✗ {gtk_signal_handler_block_by_data(2), gtk_signal_handler_unblock_by_data(2)}	2	34
✗ {error(1), isDisposed(0)}	2	31
✗ {getWarnings(0), setWarnings(1)}	2	31
✗ {getCodeGenerationSettings(1), getJavaProject(0)}	2	31
✗ {SimpleName(1), internalSetIdentifier(1)}	2	29
✗ {iterator(0), next(0)}	2	27
✓ {postValueChange(1), preValueChange(1)}	2	26
✗ {SimpleName(1), internalSetIdentifier(1)}	2	25

approach decreases when it comes to lower-ranked aspect candidates. We believe that one reason for poor performance are get and set methods that are inserted in many locations at the same time and thus out-rank actual cross-cutting concerns in the number of occurrences. Although these getters and setters are not cross-cutting, they still describe perfect usage patterns.

Furthermore, we find only few complex cross-cutting concerns. This is mainly a consequence of the condition that the locations sets have to be the same ($L_1 = L_2$). We could relax this criterion to the requirement that one location set has to be a subset of the other ($L_1 \subseteq L_2$), however, this adds exponential complexity to the determination of aspect candidates. We will improve on this in our future work. For now, let us look at three cross-cutting concerns in Eclipse.

Locking Mechanism. This cross-cutting concern was already mentioned in Section 1. Calls to both methods `lock` and `unlock` were inserted in 1 284 method locations. Here is such a location:

```
public static final native void _XFree(int address);
public static final void XFree(int /*long*/ address) {
    lock.lock();
    try {
        _XFree(address);
    } finally {
        lock.unlock();
    }
}
```

The other 1 283 method locations look similar. First `lock` is called, then a corresponding native method, and finally `unlock`. It is a typical example of a cross-cutting concern which can be easily realised using AOP.

Note that this `lock`/`unlock` concern cross-cuts different platforms. It appears in both the GTK and Motif version of Eclipse. Typically such cross-platform concerns are recognised incompletely by static and dynamic aspect mining approaches unless the platforms are analysed separately and results combined.

Abstract Syntax Trees. Eclipse represents nodes of abstract syntax trees (ASTs) by the abstract class `ASTNode` and several subclasses. These subclasses fall into the following simplified *categories*: expressions (`Expression`), statements (`Statement`), and types (`Type`). Additionally, each subclass of `ASTNode` has *properties* that cross-cut the class hierarchy. An example for a property is the *name* of a node: There are named (`QualifiedType`) and unnamed types (`PrimitiveType`), as well as named expressions (`FieldAccess`). Additional properties of a node include the *type*, *expression*, *operator*, or *body*.

This is a typical example of a *role super-imposition* concern [15]. As a result, every named subclass of `ASTNode` implements method `setName` which results in duplicated code. With AOP the concern could be realised via the method-introduction mechanism.

```
public void setName(SimpleName name) {
    if (name == null) {
        throw new IllegalArgumentException();
    }
    ASTNode oldChild = this.methodName;
    preReplaceChild(oldChild, name, NAME_PROPERTY);
    this.methodName = name;
    postReplaceChild(oldChild, name, NAME_PROPERTY);
}
```

Our mining approach revealed this cross-cutting concern with several aspect candidates. The methods `preReplaceChild` and `postReplaceChild` are called in the aforementioned `setName` method; the methods `preLazyInit` and `postLazyInit` guarantee the safe initialisation of properties; and the methods `preValueChange` and `postValueChange` are called when a new operator is set for a node.

Cloning. Another cross-cutting concern was surprising because it involved two getter methods `getStartPosition` and `getLength`. These are always called in `clone0` of subclasses of `ASTNode` and were also identified by our approach.

```
ASTNode clone0(AST target) {
    BooleanLiteral result = new BooleanLiteral(target);
    result.setSourceRange(this.getStartPosition(),
                          this.getLength());
    result.setBooleanValue(booleanValue());
    return result;
}
```

We can find complex cross-cutting concerns; once again, they are ranked on top.

7. Related Work

Previous approaches to aspect mining considered a program only at a particular time, using traditional static and dynamic program analysis techniques. One fundamental problem is their scalability. While dynamic analysis strongly depends on a compilable, executable program version and on the coverage of the used program test cases, static analyses often produce too many details and false positives as they cannot weed out non-executable code. To overcome these limitations, each approach would need additional methods which in turn make them then far less practical. Besides, many approaches require user interaction or even previous knowledge about the program.

Static Aspect Mining. The Aspect Browser [7] identifies cross-cutting concerns with textual-pattern matching (much like "grep") and highlights them. The Aspect Mining Tool (AMT) [8] combines text- and type-based analysis of source code. Ophir [19] uses a control-based comparison, applying code clone detection on program dependence graphs. Tourwé and Mens [22] introduce an identifier analysis based on formal concept analysis for mining aspectual views such as structurally related classes and methods. Krinke and Breu [12] propose an automatic static aspect mining based on control flow. The control flow graph of a program is mined for recurring execution patterns of methods. The fan-in analysis by Marin, van Deursen, and Moonen [16] determines methods that are called from many different places—thus having a high fan-in. Our approach is similar since we analyse how fan-in changed over time. In future work, we will investigate how this additional information increases precision.

Dynamic Aspect Mining. DynAMiT (Dynamic Aspect Mining Tool) [1, 3] analyses program traces reflecting the run-time behaviour of a system in search for recurring execution patterns of method relations. Tonella and Ceccato [21] suggest a technique that applies concept analysis to the relationship between execution traces and executed computational units.

Hybrid Techniques. Loughran and Rashid [14] investigate possible representations of aspects found in a legacy system in order to provide best tool support for aspect mining. Breu also reports on a hybrid approach [2] where the dynamic information of the previous DynAMiT approach is complemented with static type information such as static object types.

Mining Co-change. One of the most frequently used techniques for mining version archives is co-change. The basic idea is simple: *Two items that are changed together in the same transaction, are related to each other.* Our approach is also based on co-change. However, we use a different, more specific notion of co-change. Methods are part of a (simple) aspect candidate when they are changed together in the same transaction and *additionally the changes are the same*, i.e., a call to the same method is inserted.

Mining Co-addition of Method Calls. Recently, research extended the idea of co-change to *additions* and applied this concept to method calls: *Two method calls that are inserted together in the same transaction, are related to each other.* Williams and Hollingsworth use this observation to mine pairs of functions that form usage patterns from version archives [24]. Livshits and Zimmermann use data mining to locate patterns of arbitrary size and applied dynamic analysis to validate their patterns and identify violations [13]. Our work also investigates the addition of method calls. However, within a transaction, we do not focus on calls that are inserted together, but on locations where the same call is inserted. This allows us to identify cross-cutting concerns rather than usage patterns.

8. Conclusions and Future Work

We are the first to use version history to mine aspect candidates. The underlying hypothesis is that cross-cutting concerns emerge over time. By introducing the dimension of time, our aspect mining approach has the following advantages:

1. It *scales to industrial-sized projects* like Eclipse. In particular, we reached the highest precision (above 90%) for big projects with a long history. Additionally, we focus on concerns that cross-cut huge parts of a system. For small projects, HAM suffers from the much fewer data available, resulting in lower precision (about 60%).

2. It *discovers cross-cutting concerns across platform-specific code* (see `lock/unlock` in Section 6.5). Static and dynamic approaches recognise such concerns only when the code base is mined multiple times.

3. It *yields a high precision.* The average precision is around 50%, however, precision increases up to 90% with the project size and history.

Our work shows that version archives are indeed useful for aspect mining. Additionally, new questions arise, such as how do cross-cutting concerns evolve over time? To investigate these, we need to enhance tool support and improve the precision of our approach. Thus, our future work will concentrate on the following topics.

Tool Support. We plan to implement an Eclipse plugin that will help developers to investigate aspect candidates that were identified by HAM. The results can also be integrated into existing tools such as FEAT [18], ConcernMapper [17], or Mylar [9].

Cross-Validation. In order to improve the precision of our technique as well as to verify whether aspect candidates are present at run-time, we plan to combine HAM with DynAMiT's trace analysis [1]. We will also compare the precision of our results to existing aspect mining approaches.

Aspect Genealogy. Motivated by the work of Kim et al. [11], we plan to extend a study by Canfora and Cerulo [4] and investigate how different *classes of aspects* [15] evolve over time. Furthermore, clone detection algorithms could be run in order to detect more complex cross-cutting concerns that depend on similar code snippets.

We are hopeful that taking lessons from history will continue to help us to improve today's software.

Acknowledgements. Our work on mining software repositories was partly funded by the (German) DFG, grant Ze 509/1-1. Silvia Breu is funded by a Gates Scholarship, Thomas Zimmermann is funded by the DFG-Graduiertenkolleg "Leistungsgarantien für Rechnersysteme". Special thanks to our office inmate Christian Lindig for the fruitful discussions on the approach, and his useful comments on earlier revisions of this paper. Thanks are due to Alan Mycroft, Stefanie Scherzinger, Andreas Zeller and the anonymous reviewers for valuable comments on earlier revisions of this paper, and to Stephan Neuhaus and Valentin Dallmeier who relieved us of office work when it came to the submission deadline.

References

[1] S. Breu. Aspect Mining Using Event Traces. Master's thesis, University of Passau, Germany, March 2004.

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

[3] S. Breu and J. Krinke. Aspect Mining Using Event Traces. In *Proc. of 19th Intl. Conf. on Automated Software Engineering (ASE)*, pp. 310–315. IEEE Press, 2004.

[4] G. Canfora and L. Cerulo. How Crosscutting Concerns Evolve in JHotDraw. In *Post-Proc. of Workshop on Software Technology and Engineering Practice*, 2005.

[5] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse. How Developers Drive Software Evolution. In *Proc. of Intl. Workshop on Principles of Software Evolution (IWPSE)*. IEEE Computer Society Press, 2005.

[6] M W. Godfrey and L. Zou. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

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

[8] J. Hannemann and G. Kiczales. Overcoming the Prevalent Decomposition of Legacy Code. In *Workshop on Advanced Separation of Concerns*, 2001.

[9] M. Kersten and G. Murphy. Mylar: A Degree-of-Interest Model for IDEs. In *Proc. Intl. Conf. on Aspect-Oriented Software Development*, pp. 159–168. ACM Press, 2005.

[10] G. Kiczales et al. Aspect-Oriented Programming. In *Proc. 11th Europ. Conf. on Object-Oriented Programming*.

[11] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An Empirical Study of Code Clone Genealogies. In *Proc. of Europ. Software Engineering Conf./ACM SIGSOFT Intl. Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 187–196, New York, USA, 2005. ACM Press.

[12] J. Krinke and S. Breu. Control-Flow-Graph-Based Aspect Mining. In *1st Workshop on Aspect Reverse Engineering (WARE) at Working Conf. on Reverse Engineering*, 2004.

[13] B. Livshits and T. Zimmermann. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. In *Proc. of Europ. Software Engineering Conf./ACM SIGSOFT Intl. Symposium on the Foundations of Software Engineering*, pp. 296–305, NY, USA, 2005. ACM Press.

[14] N. Loughran and A. Rashid. Mining Aspects. In *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 2002.

[15] M. Marin, L. Moonen, and A. van Deursen. A Classification of Crosscutting Concerns. In *ICSM*, pp. 673–676. IEEE Computer Society, 2005.

[16] M. Marin, A. van Deursen, and L. Moonen. Identifying Aspects Using Fan-In Analysis. In *11th Working Conference on Reverse Engineering (WCRE)*, pp. 132–141. IEEE Computer Society, 2004.

[17] M. Robillard and F. Weigand-Warr. Concernmapper: Simple View-Based Separation of Scattered Concerns. In *Proc. of eclipse Technology eXchange (eTX) Workshop*, 2005.

[18] M. P. Robillard and G. C. Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *24th Intl. Conference on Software Engineering (ICSE)*, pp. 406–416, 2002.

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

[20] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE-21*, pp. 107–119, 1999.

[21] P. Tonella and M. Ceccato. Aspect Mining through the Formal Concept Analysis of Execution Traces. In *11th Working Conference on Reverse Engineering (WCRE)*, pp. 112–121. IEEE Computer Society, 2004.

[22] T. Tourwé and K. Mens. Mining Aspectual Views Using Formal Concept Analysis. In *Proc. of Workshop on Source Code Analysis and Manipulation (SCAM)*, pp. 97–106. IEEE Computer Society, 2004.

[23] C. C. Williams and J. K. Hollingsworth. Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480, 2005.

[24] C. C. Williams and J. K. Hollingsworth. Recovering System Specific Rules from Software Repositories. In *Proc. Intl. Workshop on Mining Software Repositories*, 2005.

[25] T. Xie and J. Pei. MAPO: Mining API Usages from Open Source Repositories. In *Proc. Intl. Workshop on Mining Software Repositories*, pp. 54–57, Shanghai, 2006.

[26] T. Zimmermann and P. Weißgerber. Preprocessing CVS Data for Fine-Grained Analysis. In *Proc. Intl. Workshop on Mining Software Repositories*, Edinburgh, 2004.