# Fast Data Mining For Programming Support

**Thomas Zimmermann**
Universität des Saarlandes, Saarbrücken, Germany
zimmerth@cs.uni-sb.de

## Abstract

We apply association rule mining to version histories: "Programmers who changed these functions also changed…". Such rules suggest and predict likely further changes and prevent errors due to incomplete changes. To deal with the required low support values we use constraints on the rules and mine on the fly. The topmost three suggestions of our ROSE prototype contain a correct location with a likelihood of 64%.

## 1 Introduction

Shopping for a book at Amazon.com, you may have come across a section that reads "Customers who bought this book also bought…", listing other books that were typically included in the same purchase. Such information is gathered by *data mining*— the automated extraction of hidden predictive information from large data sets. In this paper, we apply data mining to *version histories:* "Programmers who changed these functions also changed…". Just like the Amazon.com feature helps the customer browsing along related items, our ROSE tool guides the programmer along related changes, with the following aims:

**Suggest and predict likely changes.** Suppose a programmer has just made a change. What else does she have to change?

**Prevent errors due to incomplete changes.** A programmer has missed a change and wants to commit her work. If there are still related changes, ROSE issues a warning.

**Detect coupling undetectable by program analysis.** As ROSE operates uniquely on the version history, it is able to detect coupling between items that cannot be detected by program analysis.

ROSE is the first tool that uses fully-fledged *data mining techniques* to obtain association rules from version histories. Since ROSE requires very low support thresholds, traditional data mining techniques are too slow. Therefore, ROSE mines for rules with *constraints* to the actual user changes. These changes are not known in advance, thus the mining has to take place *on the fly*.

The remainder of this paper is organized as follows. Section 2 gives a short overview on ROSE. Section 3 shows how to gather changes and their effects from version archives, especially CVS. Section 4 describes the mining approaches for these data, including examples for association rules. In Section 5, we briefly evaluate ROSE's ability to predict future changes, based on earlier history: How often can ROSE suggest further changes, and, if so, how precise is it? Section 6 discusses related work and Section 7 closes with conclusion and consequences.

## 2 ROSE in a Nutshell

### 2.1 The "Improve Navigation" Scenario

A major application for ROSE is to guide users through source code: The user changes some entity and ROSE automatically recommends possible future changes in a view.

Figure 1 on the following page shows our ROSE tool as a plug-in for the ECLIPSE programming environment. The programmer is inserting a new preference, and has added an element to the fKeys[] array. ROSE now suggests to consider further changes, as inferred from the version history. First come the locations with the highest *confidence*—that is, the likelihood that further changes be applied to the given location.

Position 3 on the list is an HTML documentation file with a confidence of 0.75—suggesting that after adding the new preference, the documentation should be updated, too. Such a dependency is undetectable by program analysis.

### 2.2 The "Prevent Errors" Scenario

Besides supporting navigation, ROSE should also *prevent errors.* The scenario is that when a user decides to commit all her changes to the version archive, ROSE checks if there are related changes with a *high confidence* that have not been changed. If there are, like in Figure 1 the top two locations, ROSE issues a pop-up window with a warning. It also suggests the "missing" entities that should be considered (see Figure 2).

### 2.3 The Architecture of ROSE

ROSE consists of two parts:

**Preprocessing** takes a complete version archive as input. The archive is mirrored in a database (*data collection*), changes are mapped to entities and transactions (*data preparation*), and finally noise, caused by large transactions, is removed (*data cleaning*). Preprocessing ensures a fast access to all necessary information.

**Mining** creates rules from the preprocessed data. Rules describe implications between software entities, e.g., "If fKeys[] is changed, then initDefaults() is changed, too". It is possible to mine for all rules, but typically ROSE mines only for rules with a particular left-hand side. Thus, mining is speeded up and rules are always up-to-date.

More details about preprocessing and mining are described in Sections 3 and 4.

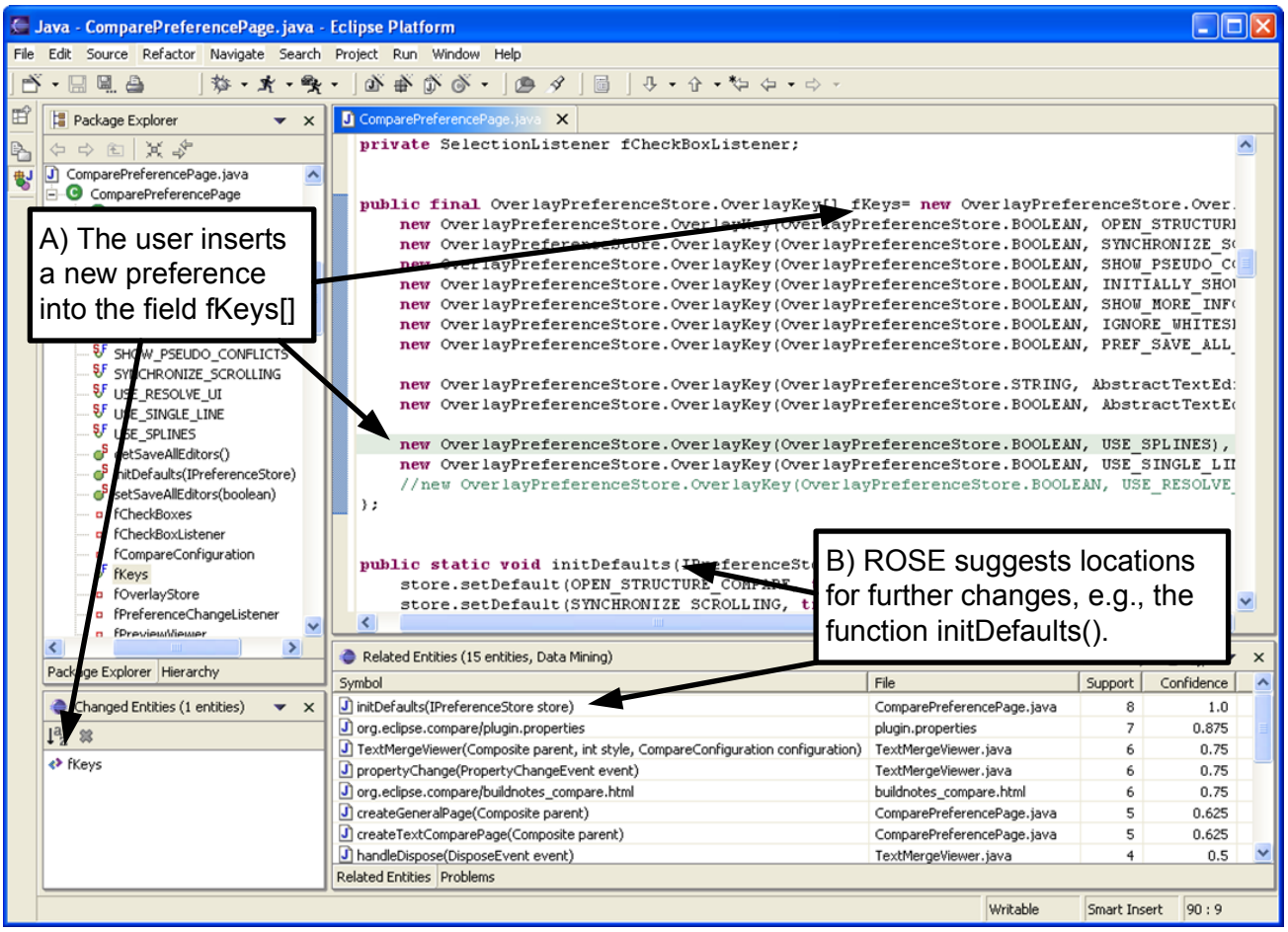


Figure 1: After the programmer has made some changes to the source (above), ROSE suggests locations (below) where, in similar transactions in the past, further changes were made.

## 3 Preprocessing

The main purpose of version control systems like CVS[1] is to store and provide different versions of files or software products. Besides versions and log information, a CVS repository contains a huge amount of additional information, e.g., what are the most frequently changed files, or what is the maximal gap between two subsequent checkins by the same author and the same log message. Unfortunately, it requires some effort, namely *data preprocessing*, to access such information—in other words to make a version control system "talkative".

Why is CVS so silent? We identified four limitations of CVS that require preprocessing:

1. *CVS has limited query functionality and is slow.*
   As mentioned above accessing information other than log information for single files is difficult in CVS. Furthermore, access is very slow because CVS uses a proprietary file format. The solution is to copy the whole CVS repository into a database. Thus, a multitude of queries are enabled and can be evaluated very fast.
2. *CVS has no atomic transactions.*
   If a developer commits several files simultaneously, CVS checks them in individually discarding the relations between them. As ROSE relies on such relations, we have to infer transactions. Usually, a transaction corresponds to exactly one commit operation.
3. *CVS knows only files—but what about functions?*
   The usefulness of ROSE depends on the granularity of its recommendations. This granularity is restricted by CVS to the file level. In order to suggest functions or declarations, we need to analyze changes and detect the affected fine-grained entities.
4. *CVS contains unreliable data.*
   Not all transactions are actually relevant for mining. For instance, merge transactions that connect two development branches are created by CVS automatically.

Many of these problems are specific to the analysis of CVS archives; more sophisticated version control systems, like SUBVERSION[2], require less data preprocessing. However, CVS is the leading version control system in open-source software development and thus provides many existing projects for test purposes.

Note that all preprocessing steps can also be done *incrementally*—it is only necessary to preprocess data for new revisions instead of working on the whole repository again. CVS provides several mechanisms to determine new revisions: One can track commits on the server-side or query for recent commits on the client-side.

The solution to the first three issues is straightforward and discussed in [20]. However, data cleaning is more sophisticated and therefore discussed in the next subsection.

[1] `http://www.cvshome.org/`

[2] `http://subversion.tigris.org/`

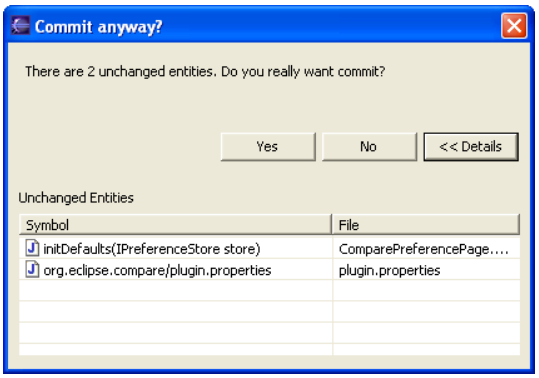


Figure 2: ROSE points programmers to locations they have likely missed.

## 3.1 Data Cleaning

Transactions that will likely induce or contribute to incorrect results are called *noise*. Such noise can contribute to new irrelevant rules or falsify existing relevant rules. ROSE performs two kinds of data cleaning:

**Explicit data cleaning** identifies noisy transactions *before* mining (during preprocessing) and tags them so that they can be ignored later on.

**Implicit data cleaning** is performed *after* mining and is based on the observation that rules induced by noisy transactions usually are weak compared to regular rules. In some cases, noise strengthens existing rules, but it never makes rules disappear. Thus, concentrating on only strong rules filters out most noisy rules.

In CVS noise evolves from several kinds of transactions:

### Large Transactions

Large transactions are very frequent and often originate from infrastructure changes. Here are two examples from OPENSSL:

- "Change #include filenames from <foo.h> *[sigh]* to <openssl.h>." *(552 files)*
- "Change functions to ANSI C." *(491 files)*

As the log messages indicate, the files contained in these transactions have been changed because of some infrastructure changes (a new compiler version), and not because of logical relations. Often such changes are performed automatically by some script.

ROSE ignores all transactions of size greater than 30 in the analysis. This bound may sound low, however, a transaction of 30 files contributes to at least $2^{30}$ association rules and increases the complexity for traditional mining algorithms dramatically.

If desired, suspect transactions can be investigated manually in order to guarantee that they are actually noise.

### Import Transactions

An import transaction contains exclusively new files and in many cases a complete subproject. Two examples taken from GCC are:

- "Initial import of libgcj" *(371 files)*
- "initial import of Java front-end" *(43 files)*

Considering such transactions for mining induces many relations between unrelated entities. It is straightforward to detect such transactions: Simply check for each transaction if all files are additions to the CVS repository. Another possible approach is to ignore additions in general.

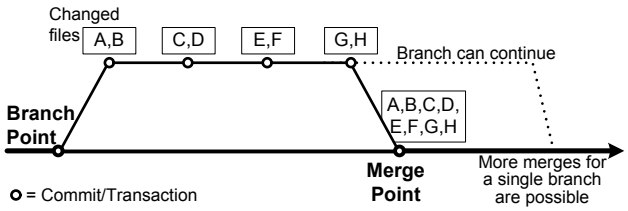


Figure 3: Merges Considered Harmful

### Merge Transactions

A more sophisticated kind of noise are merges of development branches. CVS simply reproduces all changes made from one branch to the other—in one large transaction. One example taken from GCC is

"mainline merge as of 2003-05-04" *(5874 files)*.

Figure 3 shows a smaller example. On the branch four transactions have been committed: $\{A, B\}$, $\{C, D\}$, $\{E, F\}$, and $\{G, H\}$. These files are now again changed at the merge point within a transaction that contains all changes made on the branch: $\{A, B, C, D, E, F, G, H\}$.

Merge transactions are noise for two reasons:

1. They contain unrelated changes, e.g., $B$ and $C$, thus contributing to new irrelevant rules.
2. They rank changes on branches higher (those changes are duplicated), e.g., $A$ and $B$. Thus existing rules are falsified.

Taking such transactions into account has a significant influence on the rules. Thus, transactions that resulted from merges should be identified and ignored. Unfortunately, CVS does not keep track of which revisions resulted from a merge. Michael Fischer et al. proposed a heuristic to detect these revisions [5]. However, their solution is very expensive to implement.

A simple but powerful *manual* approach is based on the observation that merges are well-documented in log messages:

1. Identify transactions whose log message contains a case-insensitive "merge".
2. Check each suspect transaction manually and verify that it is a merge transaction. This step is essential to avoid errors for log messages that incidentally contain a "merge", like the transaction "New isMerge(), isMergeWithConflicts(), and setMerge() methods".

Although this approach sounds time-consuming, the verification usually takes only a few minutes, which is nothing compared to the cost of designing and implementing an equal automatic approach.

### User-Created Noise

Detecting user-created noise, like unrelated changes within one commit, is out of reach for any approach. Although program analysis might be used, this conflicts with the goal to recognize dependencies that are undetectable by program analysis. (Program analysis would consider exactly these dependencies as noise.)

## 3.2 The Output of Preprocessing

The output of the preprocessing phase are fine-grained changes, represented by items and grouped to transactions. All these results are linked in one database table called *Lineitems* (see Figure 4 on the following page).

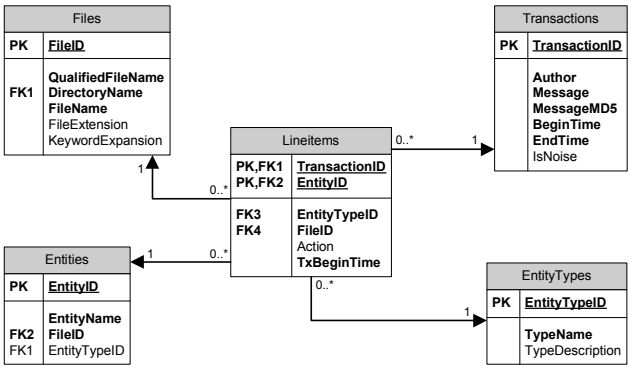


Figure 4: ROSE Database Schema: Tables for Mining

**Lineitems** contains for each transaction *TransactionID* the items, represented by Action (typically "change") and *EntityID* (a reference to the function, field or file). Additionally, the type *EntityTypeID* of each entity, the enclosing file *FileID*, and the start timestamp *TxBeginTime* of the transaction are stored. Most data of *Lineitems* is redundant in order to avoid extensive join operations.

ROSE always mines on the finest possible granularity: for source-code on *method* or *field* level, for documentation on *subsection* level, and for all other files, e.g., plugin.properties, on *file* level. Thus, only those items are inserted into table *Lineitems*. We will discuss the mining approaches in the next chapter.

# 4 Mining

## 4.1 Association Rules

Association rules represent some *pattern* in the version history. For instance, the following rule represents the pattern $\{$fKeys[], initDefaults(), plugin.properties$\}$ within the version history of ECLIPSE:

$$\text{fKeys[]} \Rightarrow \text{initDefaults()} \wedge \text{plugin.properties}$$

Such a rule has a *probabilistic* interpretation based on the *amount of evidence* in the transactions $\mathcal{D}$ they are derived from. This amount of evidence is measured by:

**Frequency and Support.** The *frequency* (or *count*) gives the number of transactions the rule has been derived from. Assume that the field fKeys[] was changed in 8 transactions. Of these 8 transactions, 7 also included changes of both the method initDefaults() and the file plugin.properties. Then, the frequency for the above rule is 7.

The *support* relates the frequency of a rule to the total number of transactions. As ECLIPSE has 44,786 transactions the support for the above rule is $7/44786 = 0.00016$.

ROSE prefers frequency to support, because a frequency of 8 is more imaginable to programmers than a confidence of 0.00016. However, support and frequency can be exchanged by each other.

**Confidence.** The *confidence* determines the certainty of the consequence if the left hand side of the rule is satisfied. In the above example, the consequence of changing initDefaults() and plugin.properties applies in 7 out of 8 transactions involving fKeys[]. Hence, the *confidence* for the above rule is $7/8 = 0.875$.

For a set of items $\mathcal{I}$, many possible rules exist: Each of the $2^{|\mathcal{I}|}$ patterns contributes to one or more rules. Thus, thresholds for support (*min_supp*) and confidence (*min_conf*) are used to reduce the number of total rules. A rule $r$ is called *strong* if and only if *support*$(r) \geq$ *min_supp* and *confidence*$(r) \geq$ *min_conf*.

## 4.2 Some Rule Examples

Let us now illustrate our approach by a few actual rules.

**Coupling in GCC.** GCC has arrays that define the cost of different assembler operations for INTEL processors. These have been changed together in 11 transactions. In 9 of these 11 transactions, this change was triggered by a change in the type *processor_cost*:

$$\begin{aligned}&(\text{i386.h}, \textit{processor\_cost})\\ \Rightarrow\ &(\text{i386.c}, \textit{i386\_cost}) \wedge (\text{i386.c}, \textit{i486\_cost}) \wedge\\ &(\text{i386.c}, \textit{k6\_cost}) \wedge (\text{i386.c}, \textit{pentium\_cost}) \wedge\\ &(\text{i386.c}, \textit{pentiumpro\_cost})\\ &\qquad [\textit{frequency} = 9; \textit{confidence} = 0.82]\end{aligned}$$

So, whenever the cost type is changed (e.g., extended for a new operation), ROSE suggests to extend the appropriate cost instances, too.[3]

**PYTHON and C files.** Our approach is not restricted to a specific programming language. In fact, we can detect coupling between program parts written in different languages (including natural language). Here is an example, taken from the PYTHON library:

$$\begin{aligned}&(\text{\_Qdmodule.c}, \textit{GrafObj\_getattr}())\\ \Rightarrow\ &(\text{qdsupport.py}, \textit{outputGetattrHook}())\\ &\quad [\textit{frequency} = 10; \textit{confidence} = 0.91]\end{aligned}$$

Whenever the C file _Qdmodule.c was changed, so was the PYTHON file qdsupport.py—a classical coupling between interface and implementation.

**POSTGRESQL documentation.** Data mining can reveal coupling between items that are not even programs, as in the POSTGRESQL documentation:

$$\begin{aligned}&\text{createuser.sgml} \wedge \text{dropuser.sgml}\\ \Rightarrow\ &\text{createdb.sgml} \wedge \text{dropdb.sgml}\\ &[\textit{frequency} = 11; \textit{confidence} = 1.0]\end{aligned}$$

Whenever both createuser.sgml and dropuser.sgml have been changed, the files createdb.sgml and dropdb.sgml have been changed, too.

## 4.3 From Rules to Recommendations

As soon as the programmer begins to make changes, the ROSE client suggests possible further changes. This is done by *applying* matching rules. In general, two notions of matching rules exist:

**Weak matching.** A rule $A \Rightarrow B$ *matches* a set of items $\Sigma$ (e.g., changed entities) if the antecedent is a subset of $\Sigma$, i.e., $A \subseteq \Sigma$.

**Strong matching.** A rule *matches* a set of items $\Sigma$ if this set is equal to the antecedent of the rule, i.e., the rule is $\Sigma \Rightarrow B$.

For both notions, the antecedent of a rule is satisfied, but only for strong matching it is satisfied exactly. We refer to

[3] This rule also holds for the other direction, with the same frequency and (incidentally) the same confidence.

the set of items $\Sigma$ as the *situation* in which ROSE makes recommendations.

Considering weak matching rules for recommendations is not reasonable because this bypasses support and confidence thresholds. Suppose that we have three functions f(), g(), and h(). The functions g() and h() exclude each other. Thus, no strong rule f() $\wedge$ g() $\Rightarrow$ h() exists because it has no support. The user changes f() and g(). Using weak matching, we would consider the rule f() $\Rightarrow$ h() and falsely recommend h() —which is excluded by the occurrence of g(). As ROSE uses *strong* rules for recommendations, it also has to use *strong* matching.

**How does ROSE compute suggestions?**
The set of suggestions for a situation $\Sigma$ and a set of strong rules $\mathcal{R}$ is defined as the *union* of the consequents of all matching rules:

$$\mathit{apply}_{\mathcal{R}}(\Sigma) = \bigcup_{(\Sigma \Rightarrow B) \in \mathcal{R}} B$$

## 4.4 The Apriori Approach for Mining Association Rules

One of the most popular approaches for mining *all strong* association rules is the Apriori algorithm [1; 12]. It takes a *min_supp* and a *min_conf* threshold and the task-relevant data $\mathcal{D}$ as input.

Internally, the Apriori algorithm represents patterns with *itemsets*. A $k$-itemset is an itemset of size $k$. An itemset is called *frequent* if it satisfies the support (or frequency) threshold. The set of all frequent $k$-itemsets is denoted as $L_k$. The *Apriori property* helps to reduce the search space for frequent itemsets:

**All nonempty subsets of a frequent itemset must also be frequent.**

This is obvious because the support increases, if items $X$ are removed from an itemset $I \subseteq \mathcal{I}$: $P(I) \leq P(I - X)$. Thus, if $I$ was frequent, *min_supp* $\leq P(I)$, then $I - X$ is frequent, too: *min_supp* $\leq P(I) \leq P(I - X)$.

The Apriori algorithm consists of two phases:

1. **Find all frequent itemsets.**
   Frequent itemsets are generated level-wise: First $L_1$ is computed, then $L_1$ is used to find $L_2$ which is used to compute $L_3$, and so on. This phase terminates if for a $k$ no more frequent $k$-itemsets are found. Each level, i.e., the creation of a set $L_k$, consists of four steps:
   - The *join* step:
     A candidate $k$-itemset $C_k$ is generated by joining $L_{k-1}$ with itself. The join condition is that the first $k-1$ items of two itemsets $l_1$ and $l_2$ are equal and only the last elements differ: $l_1[k] < l_2[k]$.
   - The *prune* step:
     Remove itemsets from $C_k$ that cannot be frequent by means of the Apriori property.
   - The *scan* or *count* step:
     Scan the database $\mathcal{D}$ and count the frequency of each remaining candidate in $C_k$.
   - The *create* step:
     The frequent $k$-itemsets $L_k$ are those sets in $C_k$ that satisfy the frequency threshold.

   Searching for frequent itemsets is the most time consuming part of the Apriori algorithm; each level requires a full scan of the database. Thus, the support (or frequency) threshold has a huge impact on running time.

2. **Generate association rules from frequent itemsets.**

   For each frequent itemset $l$ all nonempty subsets $s$ are created. Such a subset results in a rule $s \Rightarrow l - s$ if and only if:

   $$\mathit{confidence}(s \Rightarrow l - s) = P(l - s \mid s) \geq \mathit{min\_conf}$$

   The test for the support (or frequency) threshold can be omitted because rules are created from frequent itemsets. Therefore the following test is always true:

   $$\mathit{support}(s \Rightarrow l - s) = P(l - s \cup s) = P(l) \geq \mathit{min\_supp}$$

Figure 5 on the next page shows an example for the Apriori algorithm. The candidate 1-itemset $C_1$ corresponds to the set of all items $\mathcal{I}$. The count step reveals that itemset $\{E\}$ is not frequent. Next, the candidate 2-itemsets $C_2$ are generated by joining $L_1$ with itself ($C_2$ is always the cross product of $L_1$). For $k = 2$ it is never possible to prune any elements because all subsets are singletons and always contained in $L_1$. The count step identifies $\{B, D\}$ and $\{C, D\}$ as not frequent. Next, the candidate 3-itemsets $C_3$ are generated from $L_2$ using the join condition $l_1[1] = l_2[1] \wedge l_1[2] < l_2[2]$. This returns three itemsets. Two of them are not frequent by the Apriori property and pruned: For $\{A, B, D\}$ the subset $\{B, D\}$ is not frequent and for the itemset $\{A, C, D\}$ subset $\{C, D\}$ is not frequent. For the third candidate $\{A, B, C\}$ a database scan verified that it is frequent. After all frequent itemsets have been computed, each itemset in $L_2$ and $L_3$ is used to create rules. The confidence is computed for each rule and only strong rules are returned.

Keep in mind, that the Apriori property can only tell that an itemset is *not* frequent. A check for an itemset *being* frequent always has to scan the database.

The Apriori algorithm has several drawbacks: The database $\mathcal{D}$ is repeatedly scanned for each level of the frequent itemset creation. Additionally, the creation of candidate sets is expensive. If there are $10^4$ frequent 1-itemsets then about $10^8$ candidate 2-itemsets are generated. Moreover, to discover a pattern of size 100, the Apriori algorithm must create more than $2^{100}$ candidates in total.

It is possible to mine association rules without candidate generation based on a divide-and-conquer strategy. The algorithm is called *frequent-pattern growth* and also known as *FP-growth* [8].

## 4.5 The ROSE Approach for Mining Association Rules

The classical use of the Apriori algorithm is to compute all rules above a minimum support and confidence. However, computing all rules is useful for searching general patterns but not for making recommendations:

**The coverage of Apriori is too low.** The *coverage* is directly proportional to the number of distinct antecedents within a rule set $\mathcal{R}$. A high coverage is desirable because ROSE can then make recommendations in most cases. A low coverage means that ROSE is often clueless.

The coverage can be increased by extending the rule set $\mathcal{R}$, e.g., by *lowering* the confidence and especially *the support thresholds*. However, for too low support thresholds Apriori may take months. The bottleneck is not Apriori but the circumstance that $\mathcal{R}$ gets too large—greater than $2^{|\mathcal{I}|}$ in worst case.

**Transactions:** $\mathcal{D}$

| TxID | List of items |
|---|---|
| 100 | A, B, C |
| 200 | A, D |
| 300 | A, B, C |
| 400 | B, D |
| 500 | A, D |
| 600 | B, E |
| 700 | A, B |

**Generate frequent 1-itemset $L_1$**

$C_1$

| Itemset | Count |
|---|---|
| $\{A\}$ | 5 |
| $\{B\}$ | 5 |
| $\{C\}$ | 2 |
| $\{D\}$ | 3 |
| $\{E\}$ | 1 |

count →

create →

$L_1$

| Itemset | Count |
|---|---|
| $\{A\}$ | 5 |
| $\{B\}$ | 5 |
| $\{C\}$ | 2 |
| $\{D\}$ | 3 |

**Generate frequent 2-itemset $L_2$**

join → $C_2$

| Itemset |
|---|
| $\{A, B\}$ |
| $\{A, C\}$ |
| $\{A, D\}$ |
| $\{B, C\}$ |
| $\{B, D\}$ |
| $\{C, D\}$ |

prune →

$C_2$

| Itemset |
|---|
| $\{A, B\}$ |
| $\{A, C\}$ |
| $\{A, D\}$ |
| $\{B, C\}$ |
| $\{B, D\}$ |
| $\{C, D\}$ |

count →

$C_2$

| Itemset | Count |
|---|---|
| $\{A, B\}$ | 3 |
| $\{A, C\}$ | 2 |
| $\{A, D\}$ | 2 |
| $\{B, C\}$ | 2 |
| $\{B, D\}$ | 1 |
| $\{C, D\}$ | 0 |

create →

$L_2$

| Itemset | Count |
|---|---|
| $\{A, B\}$ | 3 |
| $\{A, C\}$ | 2 |
| $\{A, D\}$ | 2 |
| $\{B, C\}$ | 2 |

**Generate frequent 3-itemset $L_3$**

join → $C_3$

| Itemset |
|---|
| $\{A, B, C\}$ |
| $\{A, B, D\}$ |
| $\{A, C, D\}$ |

prune →

$C_3$

| Itemset |
|---|
| $\{A, B, C\}$ |

count →

$C_3$

| Itemset | Count |
|---|---|
| $\{A, B, C\}$ | 2 |

create →

$L_3$

| Itemset | Count |
|---|---|
| $\{A, B, C\}$ | 2 |

**Generate association rules from $L_2$ and $L_3$**

| Frequent itemset | Rule | Confidence | Strong |
|---|---|---|---|
| $\{A, B\}$ | $A \Rightarrow B$ | $3/5 = 0.60$ | yes |
| | $B \Rightarrow A$ | $3/5 = 0.60$ | yes |
| $\{A, C\}$ | $A \Rightarrow C$ | $2/5 = 0.40$ | no |
| | $C \Rightarrow A$ | $2/2 = 1.00$ | yes |
| $\{A, D\}$ | $A \Rightarrow D$ | $2/5 = 0.40$ | no |
| | $D \Rightarrow A$ | $2/3 = 0.67$ | yes |
| $\{B, C\}$ | $B \Rightarrow C$ | $2/5 = 0.40$ | no |
| | $C \Rightarrow B$ | $2/2 = 1.00$ | yes |
| $\{A, B, C\}$ | $A \Rightarrow B \wedge C$ | $2/5 = 0.40$ | no |
| | $B \Rightarrow A \wedge C$ | $2/5 = 0.40$ | no |
| | $C \Rightarrow A \wedge B$ | $2/2 = 1.00$ | yes |
| | $A \wedge B \Rightarrow C$ | $2/3 = 0.67$ | yes |
| | $A \wedge C \Rightarrow B$ | $2/2 = 1.00$ | yes |
| | $B \wedge C \Rightarrow A$ | $2/2 = 1.00$ | yes |

Figure 5: An Example for the Apriori Algorithm (*min_freq* $= 2$; *min_conf* $= 0.5$)

Of course, too low support thresholds have a bad influence on the quality of recommendations. Nevertheless, the developer should be able to decide on support thresholds independently from any technical boundaries imposed by the Apriori algorithm.

**The search for matching rules is expensive.** As mentioned above, $\mathcal{R}$ gets very large—for most projects a multiple of the number of transactions. Thus, the search for matching rules is expensive if $\mathcal{R}$ does not fit into memory and no suitable index structures are available.

Therefore, ROSE uses its own mining algorithm that mines *only required* rules *on the fly*. This algorithm is based on two optimizations:

**Mine with constrained antecedents.** In our specific case, the antecedent is equal to the situation; hence, we only mine rules *on the fly* which *match* the situation $\Sigma$, i.e., rules that are $\Sigma \Rightarrow I$. Mining with such constrained antecedents takes only a few seconds. An additional advantage of this approach is that it is incremental in the sense that it allows new transactions to be added to $\mathcal{D}$ between two situations. Thus, recommendations are always up-to-date.

**Mine only single-consequents.** To speed up the mining process even more, we only compute rules with a single item in their consequent. So, for a situation $\Sigma$, the rules have the form $\Sigma \Rightarrow \{i\}$. For ROSE, such rules are sufficient because ROSE computes the union of the consequents anyway. Therefore, considering multi-consequents is superfluous: For each item $i \in I$ of a multi-consequent rule $\Sigma \Rightarrow I\ [s; c]$ exists a single-consequent rule $\Sigma \Rightarrow \{i\}\ [s_i; c_i]$ with higher or equal support and confidence values $s_i \geq s$ and $c_i \geq c$ because *frequency*$(\Sigma \cup \{i\}) \geq$ *frequency*$(\Sigma \cup B)$. Thus, for ROSE single-consequent rules have the same expressive power as multi-consequent rules.

The ROSE mining algorithm consists of three steps:

1. **Find relevant transactions.**
   Find the set of all transactions $\mathcal{T}$ that contain *all* items of the situation $\Sigma$, i.e., $\mathcal{T} = \{T \mid T \in \mathcal{D}, \Sigma \subseteq T\}$.
2. **Generate relevant frequent itemsets.**
   Group the items of these transactions (*Lineitems* $\bowtie \mathcal{T}$) by their entities (identified by *EntityID*), and sort them by their descending count.
3. **Create single-consequent rules.**
   Each group corresponds to exactly one single-consequent rule.
   - The *frequency* of $\Sigma$ is the maximal count of a group (which is likely for an item $i \in \Sigma$ and is always for the count of the first returned group).
   - The *frequency* for the rule $\Sigma \Rightarrow \{i\}$ corresponds to the count for group of the item $i$.
   - The *confidence* of a rule $\Sigma \Rightarrow \{i\}$ is
   $$\frac{\textit{frequency}(\Sigma \Rightarrow \{i\})}{\textit{frequency}(\Sigma)}$$
   - Ignore *trivial* rules—that are rules $\Sigma \Rightarrow \{i\}$ with $i \in \Sigma$.

   Return only rules that satisfy the support and confidence thresholds (which means they are strong).

**Situation** $\Sigma = \{A, B\}$ **and** $k = |\ \Sigma\ | = 2$

**Find relevant transactions** $\mathcal{T}$

| TxID | List of items |
|---|---|
| 100 | A, B, C |
| 200 | A, D |
| 300 | A, B, C |
| 400 | B, D |
| 500 | A, D |
| 600 | B, E |
| 700 | A, B |

find →

Relevant transactions $\mathcal{T}$

| TxID | List of items |
|---|---|
| 100 | A, B, C |
| 300 | A, B, C |
| 700 | A, B |

**Generate frequent** $k$**- and** $k+1$**-itemsets that contain** $\Sigma$

generate →

| Item | Frequency | Itemset |
|---|---|---|
| A | 3 | ⇒ {A, B } |
| B | 3 | ⇒ {A, B } |
| C | 2 | ⇒ {A, B, C } |

**Create single-consequent rules with antecedent** $\Sigma$

| Item | Frequency | Rule |
|---|---|---|
| A | *frequency*($\Sigma$) =3 | {A,B} ⇒ {A} is trivial |
| B | 3 | {A,B} ⇒ {B} is trivial |
| C | 2 | {A,B} ⇒ {C} is strong |

Figure 6: An Example for the ROSE Algorithm (*min_freq* $= 2$; *min_conf* $= 0.5$)

Figure 6 shows an example for the ROSE mining algorithm. Suppose, the situation is $\Sigma = \{A, B\}$. First, ROSE searches all transactions that contain $\Sigma$: 100, 300, and 700. Next, it groups exactly those transactions by items and sorts them by their descending count. The highest count is for item $A$, thus the *frequency* for $\Sigma$ is 3. The rules for $A$ and $B$ are trivial (both are in the situation $\Sigma$), thus they are ignored. For $C$, the rule $\Sigma = \{A, B\} \Rightarrow \{C\}$ is strong because the thresholds for *min_freq* and *min_conf* are satisfied.

The optimizations above make mining very efficient: The average runtime of a query is about 0.5s for large version histories like GCC with more than 45,000 transactions.[4]

### 4.6 Binary Association Rules

ROSE provides another mining algorithm for *single-antecedent single-consequent* rules $\{a\} \Rightarrow \{b\}$. Such rules are less precise for recommendations, but valuable for measurement and visualization of coupling between entities [19]. The algorithm is exactly like the Apriori algorithm presented in Subsection 4.4, except that only frequent 2-itemsets are generated and used for rule creation.

## 5 Evaluation

In [21] we evaluated the predictiveness of association rules for programming support. This section summarizes some of the results for the following issues:

**Navigation through source code.** Given a single changed entity, can ROSE point programmers to entities that should typically be changed, too?

**Error prevention.** Can ROSE prevent errors? Say, the programmer has changed many entities but has missed to change one entity. Does ROSE find the missing one?

[4]Measured on a PC Intel 2.0 GHz Pentium 4 with 1 GB RAM.

| Project, Description | Transactions evaluated | Transactions total |
|---|---|---|
| ECLIPSE, integrated environment | 2,965 | 46,843 |
| GCC, compiler collection | 1,083 | 47,424 |
| GIMP, image manipulation tool | 1,305 | 9,796 |
| JBOSS, application server | 1,320 | 10,843 |
| JEDIT, text editor | 577 | 2,024 |
| KOFFICE, office suite | 1,385 | 20,903 |
| POSTGRESQL, database system | 925 | 13,477 |
| PYTHON, language + library | 1,201 | 29,588 |

Table 1: Analyzed projects

**Closure.** Suppose a transaction is finished—the programmer made all necessary changes. How often does ROSE erroneously suggest that a change is missing?

### 5.1 Evaluation Setup

For our evaluation, we analyzed the archives of eight large open-source projects (Table 1). For each archive, we chose a number of full months containing the last 1,000 transactions, but not more than 50% of all transactions as our *evaluation period*. In this period, we check for each transaction $T$ whether its items can be *predicted from earlier history:*

1. We create a *test case* $q = (Q, E)$ consisting of a *query* $Q \subset T$ and an *expected outcome* $E = T - Q$.
2. We take all transactions $T_i$ that have been completed before *time*$(T)$ as a *training set* and mine a set of rules $R$ from these transactions.
3. To avoid having the user work through endless lists of suggestions, ROSE only shows the *top ten single-consequent rules* $R_{10} \subset R$ ranked by confidence. In our evaluation, we apply $R_{10}$ to get the result of the query $A_q = \mathit{apply}_{R_{10}}(Q)$. So, the size of $A_q$ is always less or equal than ten.
4. The result $A_q$ of a test case $q$ consists of two parts:
   - $A_q \cap E_q$ are the items that *matched* the expected outcome and are considered *correct* predictions
   - $A_q - E_q$ are unexpected recommendations that are considered as *wrong* predictions and called *false positives*.

   Additionally, ROSE may have missed items:
   - $E_q - A_q$ are the *missing* predictions and called *false negatives*.

For the assessment of a result $A_q$, we use two measures from information retrieval [15]: The *precision* $P_q$ describes which fraction of the returned items was actually expected by the user. The *recall* $R_q$ indicates the percentage of expected items that were returned.

$$P_q = \frac{|A_q \cap E_q|}{|A_q|} \qquad R_q = \frac{|A_q \cap E_q|}{|E_q|}$$

In case no items are returned ($A_q$ is empty), we define the precision as $P_q = 1$, and in case no items are expected, we define the recall as $R_q = 1$.

For each query $q_i$, we get a precision-recall pair $(P_{q_i}, R_{q_i})$. We use *micro-evaluation* to summarize these pairs into a single average precision-recall pair:

$$P_\mu = \frac{\sum_{i=1}^{N} |A_{q_i} \cap E_{q_i}|}{\sum_{i=1}^{N} |A_{q_i}|} \quad R_\mu = \frac{\sum_{i=1}^{N} |A_{q_i} \cap E_{q_i}|}{\sum_{i=1}^{N} |E_{q_i}|}$$

One can think of micro-evaluation as summarizing all queries into one large query and then computing precision and recall for this large query. It therefore allows statements on *single suggestions* like "every $n$th suggestion is wrong/correct". For example, the precision $P_\mu$ for PYTHON is 0.50: Every second suggestion is correct, which means that the recommended entity was actually changed later on. In contrast, macro-evaluation makes statements per query.

Keep in mind that ROSE usually provides several suggestions for a query. In order to assess the actual usefulness for the programmer, we thus checked the *likelihood* whether the expected location would be included in ROSE's *top three* navigation suggestions (assuming that a programmer won't have too much trouble judging the first three suggestions).

Formally, $L_3$ is the likelihood that for a query $q = (Q, E)$, at least one of the first three recommendations is correct:

$$L_3 = L(|\mathit{apply}(Q, R_3) \cap E| > 0)$$

where $L(p)$ stands for the probability of the predicate $p$.

*One can either have **precise** suggestions or **many** suggestions, but not both.*

### 5.2 Results: Improve Navigation

We evaluated the predictive power of ROSE in the "Improve Navigation" scenario. For each transaction $T$, and each item $i \in T$, we queried $Q = \{i\}$, and checked whether ROSE would predict $E = T - \{i\}$. For each transaction, we thus tested $|T|$ queries, each with one element.

The results for *min_freq* $= 1$ and *min_conf* $= 0.1$ are shown in Table 2 (column *Navigation*):

- The average *recall* of 0.15 means that ROSE's suggestion correctly included 15% of all changes that were actually carried out in the given transactions.
- The average *precision* of 0.26 means that 26% of all recommendations were correct—every fourth suggested change was actually carried out (and thus predicted correctly by ROSE).
- The average *likelihood* $L_3$ of the three topmost suggestions predicting a correct location is 0.64. This means for 64% of all transactions one of ROSE's predictions took place.

While KOFFICE and JEDIT have lower recall, precision, and likelihood values, GCC strikes by overall high values. The reason is that KOFFICE and JEDIT are projects where continuously many new features are inserted (which cannot be predicted from history) while GCC is a stable system where the focus is on maintaining existing features.

*When given one initial changed entity, ROSE can predict 15% of all entities changed later in the same transaction. In 64% of all transactions, ROSE's topmost three suggestions contain a correct location.*

### 5.3 Results: Error Prevention

We determined in how many cases ROSE can predict a missing entity in the "Error Prevention" scenario. For this purpose, we took each transaction, left out one item and checked if ROSE could predict the missing item. In other words, the query was the complete transaction without the missing item. So, for each single transaction $T$, and each entity $i \in T$, we queried $Q = T - \{i\}$, and checked whether ROSE would predict $E = \{i\}$. For each transaction, we thus again ran $|T|$ tests.

| | Navigation | | | Prevention | | Closure |
|---|---|---|---|---|---|---|
| Frequency | 1 | | | 3 | | 3 |
| Confidence | 0.1 | | | 0.9 | | 0.9 |
| Project | $R_\mu$ | $P_\mu$ | $L_3$ | $R_\mu$ | $P_\mu$ | $P_M$ |
| ECLIPSE | 0.15 | 0.26 | 0.53 | 0.02 | 0.48 | 0.979 |
| GCC | 0.28 | 0.39 | 0.89 | 0.20 | 0.81 | 0.953 |
| GIMP | 0.12 | 0.25 | 0.91 | 0.03 | 0.71 | 0.978 |
| JBOSS | 0.16 | 0.38 | 0.69 | 0.01 | 0.24 | 0.981 |
| JEDIT | 0.07 | 0.16 | 0.52 | 0.004 | 0.59 | 0.986 |
| KOFFICE | 0.08 | 0.17 | 0.46 | 0.003 | 0.24 | 0.990 |
| POSTGRES | 0.13 | 0.23 | 0.59 | 0.03 | 0.66 | 0.989 |
| PYTHON | 0.14 | 0.24 | 0.51 | 0.01 | 0.50 | 0.986 |
| Average | 0.15 | 0.26 | 0.64 | 0.04 | 0.50 | 0.980 |

Table 2: Results ($R$ = recall; $P$ = precision; $L$ = likelihood)

As too many false warnings might undermine ROSE's credibility, ROSE is set up to issue warnings only if the *high confidence threshold* of 0.9 is exceeded. The results are shown in Table 2 (column *Prevention*):

- The average *recall* is about 4%. This means that in only one out of 25 queries (in GCC: every 5th query), ROSE correctly predicted the missing item.
- The average *precision* is above 50%. This means that if a warning occurs, every second recommendation of ROSE is correct.

*Given a transaction where one change is missing, ROSE can predict 4% of the entities that need to be changed. On average, every second recommended entity is correct.*

### 5.4 Results: Closure

The final question in the "Error Prevention" scenario is how many false alarms ROSE would produce in case no entity is missing. We simulated this by testing *complete transactions*. For each transaction $T$, we queried $Q = T$, and checked whether ROSE would predict $E = \emptyset$; we thus had one test per transaction.

We measured the percentage $P_M$ of transactions where ROSE has not issued a warning.[5] Thus $1 - P_M$ is the percentage of false alarms.

The results are shown in Table 2 (column *Closure*). One can see that the precision is very high for all projects, usually around 0.98. This means that ROSE issues a false alarm in only every 50th transaction.

*ROSE's warnings about missing changes should be taken seriously: Only 2% of all transactions cause a false alarm. In other words: ROSE does not stand in the way.*

## 6 Related Work

Independently from us, Annie Ying developed an approach that also uses association rule mining on CVS version archives [18]. She especially evaluated the usefulness of the results, considering a recommendation most valuable or "surprising" if it could not be determined by program analysis, and found several such recommendations in the MOZILLA and ECLIPSE projects. In contrast to ROSE,

[5] In this case the percentage corresponds to the precision of *macro*-evaluation. Therefore we denote it as $P_M$.

though, Ying's tool can only suggest files, not finer-grained entities, and does not support mining-on-the-fly.

Gall et al. were the first to use release data to detect logical coupling between modules [6]. The CVS history allows to detect more fine-grained logical coupling between classes [7], files and functions [19]. None of these works on logical coupling did address its predictive power.

Jelber Sayyad-Shirabad et al. use inductive learning to learn different concepts of relevance between logically coupled files [16; 17]. A concept is a relevance relation, for example whether two files have been updated simultaneously. Instances of concepts are described in terms of *attributes* such as file name, extension and simple metrics like number of routines defined. Jelber Sayyad-Shirabad thoroughly evaluated the predictive power of the concepts found, but none of the papers give a convincing example of such a concept.

Amir Michail used data mining on the source code of programming libraries to detect reuse patterns in form of association [13] or generalized association rules [14]. The latter takes inheritance relations into account. The items in these rules are (re-)use relationships like method invocation, inheritance, instantiation, or overriding. Both papers lack an evaluation of the quality of the patterns found. However, Michail mines a single version, while ROSE uses the changes between different versions.

To guide programmers, a number of tools have exploited *textual similarity* of log messages [3] or program code [2]. HIPIKAT [4] improves on this by taking also other sources like mail archives and online documentation into account. In contrast to ROSE, all these tools focus on high recall rather than on high precision, and on relationships between files or classes rather than between fine-grained entities.

## 7 Conclusion and Consequences

ROSE can be a helpful tool in suggesting further changes to be made, and in warning about missing changes. But the more there is to learn from history, the more and better suggestions can be made:

- For stable systems like GCC, ROSE gives many and precise suggestions: 28% of related entities can be predicted, with a precision of about 40% for each single suggestion, and a likelihood of over 90% for the three topmost suggestions.
- For rapidly evolving systems like KOFFICE or JEDIT, ROSE's most useful suggestions are at the file level. Overall, this is not surprising, as ROSE would have to predict *new functions*—which is probably out of reach for any approach.
- In about 4% of all erroneous transactions, ROSE correctly detects the missing change. If such a warning occurs, it should be taken seriously, as only 2% of all transactions cause false alarms.

What have *we* learned from history, and what are our suggestions? Here are our plans for future work:

**Taxonomies.** Every change in a method implies a change in the enclosing class, which again implies changes in the enclosing files or packages. We want to exploit such *taxonomies* to identify patterns such as "this change implies a change in this package" (rather than "in this method") that may be less precise in the location, but provide higher confidence.

**Sequence rules.** Right now, we are only relating changes that occur in the *same* transaction. In the future, we also want to detect rules across multiple transactions: "The system is always tested before being released" (as indicated by appropriate changes).

**Further data sources.** Archived changes contain more than just author, date, and location. One could scan *log messages* (including the one of the change to be committed) to determine the concern the change is more likely to be related to (say, "Fix" vs. "New feature").

We are currently making ROSE available as a plug-in for ECLIPSE. For more information and download, visit

`http://www.st.cs.uni-sb.de/softevo/`

**Acknowledgments.** This project is funded by the Deutsche Forschungsgemeinschaft, grant Ze 509/1-1. Special thanks to the project members Michael Burch, Stephan Diehl, Peter Weißgerber, and Andreas Zeller for their support. Olaf Herden gave helpful comments on earlier revisions of this paper.

## References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th Very Large Data Bases Conference (VLDB)*, pages 487–499. Morgan Kaufmann, 1994.

[2] D. L. Atkins. Version sensitive editing: Change history as a programming tool. In B. Magnusson, editor, *Proceedings of System Configuration Management SCM'98*, volume 1439 of *LNCS*, pages 146–157. Springer-Verlag, 1998.

[3] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail. CVSSearch: searching through source code using CVS comments. In ICSM 2001 [9], pages 364–374.

[4] D. Čubranić and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proc. 25th International Conference on Software Engineering (ICSE)*, pages 408–418, Portland, Oregon, May 2003.

[5] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In ICSM 2003 [10].

[6] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proc. International Conference on Software Maintenance (ICSM '98)*, pages 190–198, Washington D.C., USA, Nov. 1998. IEEE.

[7] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In IWPSE 2003 [11], pages 13–23.

[8] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM Press, 2000.

[9] *Proc. International Conference on Software Maintenance (ICSM 2001)*, Florence, Italy, Nov. 2001. IEEE.

[10] *Proc. International Conference on Software Maintenance (ICSM 2003)*, Amsterdam, Netherlands, Sept. 2003. IEEE.

[11] *Proc. International Workshop on Principles of Software Evolution (IWPSE 2003)*, Helsinki, Finland, Sept. 2003. IEEE Press.

[12] H. Mannila, H. Toivonen, and A. I. Verkamo. Efficient algorithms for discovering association rules. In U. M. Fayyad and R. Uthurusamy, editors, *AAAI Workshop on Knowledge Discovery in Databases (KDD-94)*, pages 181–192, July 1994.

[13] A. Michail. Data mining library reuse patterns in user-selected applications. In *Proc. 14th International Conference on Automated Software Engineering (ASE'99)*, pages 24–33, Cocoa Beach, Florida, USA, Oct. 1999. IEEE Press.

[14] A. Michail. Data mining library reuse patterns using generalized association rules. In *International Conference on Software Engineering*, pages 167–176, 2000.

[15] C. J. V. Rijsbergen. *Information Retrieval, 2nd edition*. Butterworths, London, 1979.

[16] J. Sayyad-Shirabad, T. C. Lethbridge, and S. Matwin. Supporting maintainance of legacy software with data mining techniques. In ICSM 2001 [9], pages 22–31.

[17] J. Sayyad-Shirabad, T. C. Lethbridge, and S. Matwin. Mining the maintenance history of a legacy software system. In ICSM 2003 [10].

[18] A. T. T. Ying. Predicting source code changes by mining revision history. Master's thesis, University of British Columbia, Canada, Oct. 2003.

[19] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In IWPSE 2003 [11], pages 73–83.

[20] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, pages 2–6, Edinburgh, Scotland, UK, May 2004.

[21] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proc. 26th International Conference on Software Engineering (ICSE)*, Edinburgh, Scotland, May 2004.