Mining Version Histories to Guide Software Changes

Thomas Zimmermann tz@acm.org
Peter Weißgerber weissger@st.cs.uni-sb.de
Stephan Diehl diehl@acm.org
Andreas Zeller zeller@acm.org

Saarland University, Saarbrücken, Germany

Abstract

We apply data mining to version histories in order to guide programmers along related changes: "Programmers who changed these functions also changed…". Given a set of existing changes, such rules (a) suggest and predict likely further changes, (b) show up item coupling that is indetectable by program analysis, and (c) prevent errors due to incomplete changes. After an initial change, our ROSE prototype can correctly predict 26% of further files to be changed—and 15% of the precise functions or variables. The topmost three suggestions 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 you are a programmer and just made a change. What else do you have to change? Figure 1 on the following page shows our ROSE tool as a plug-in for the ECLIPSE programming environment. The programmer is extending ECLIPSE itself with a new preference, and has added an element to the fKeys[] array. ROSE now suggests to consider further changes, as inferred from the ECLIPSE version history. First come the locations with the highest *confidence*—that is, the likelihood that further changes be applied to the given location.

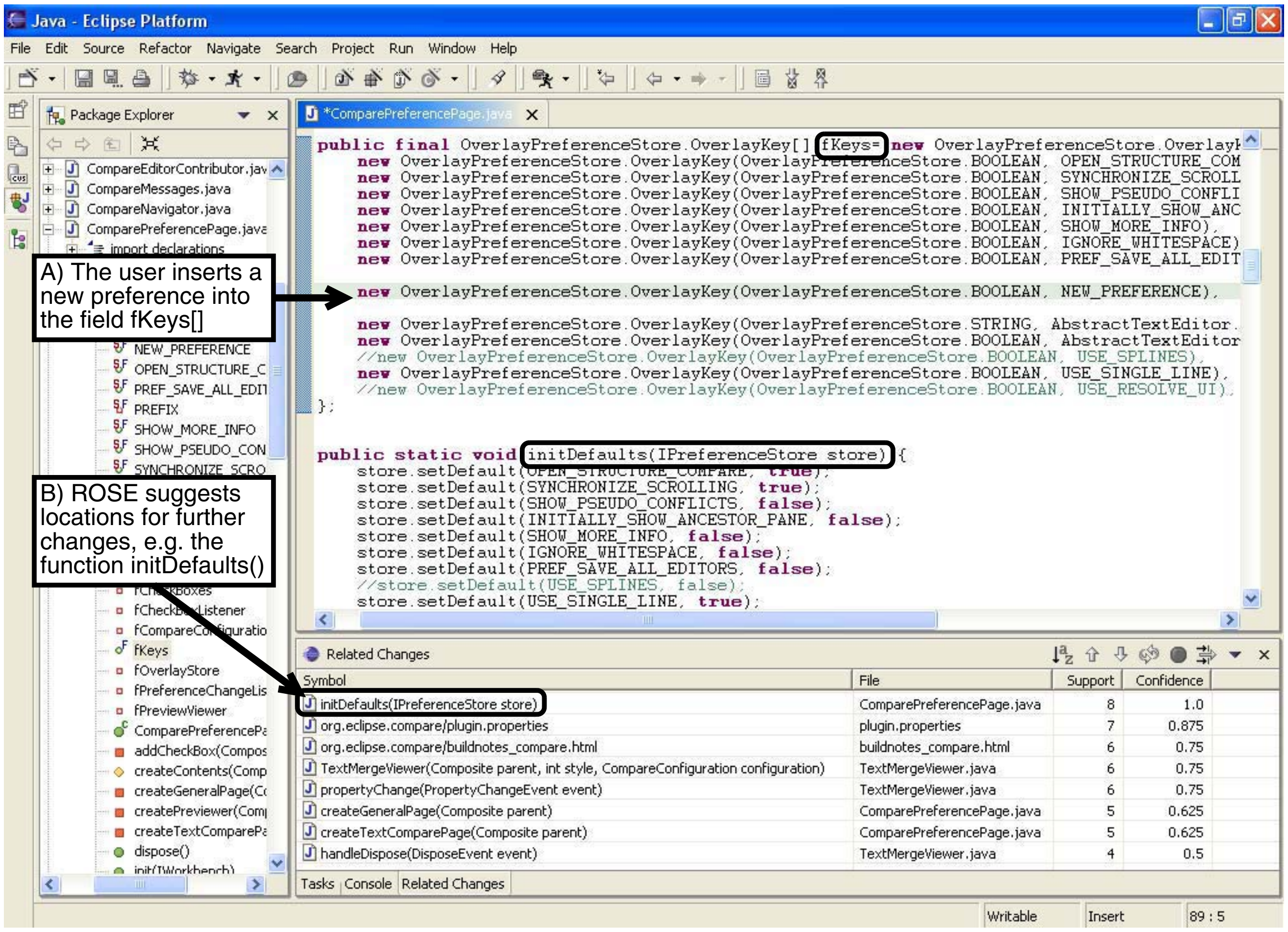
Prevent errors due to incomplete changes. In Figure 1, the top location has a confidence of 1.0: In the past, each time some programmer extended the fKeys[] array, she also extended the function that sets the preference default values. If the programmer now wanted to commit her changes *without* altering the suggested location, ROSE would issue a warning.

Detect coupling indetectable 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—including coupling between items that are not even programs. In Figure 1, position 3 on the list is an ECLIPSE HTML documentation file with a confidence of 0.75—suggesting that after adding the new preference, the documentation should be updated, too.

ROSE is not the first tool to leverage version histories. In earlier work (Section 7), researchers have used history data to understand programs and their evolution [3], to detect evolutionary coupling between files [8] or classes [4], or to support navigation in the source code [6]. In contrast to this state of the art, the present work

- uses full-fledged *data mining techniques* to obtain association rules from version histories,
- detects coupling between fine-grained *program entities* such as functions or variables (rather than, say, classes), thus increasing precision and integrating with program analysis,
- thoroughly evaluates the *ability to predict future or missing changes,* thus evaluating the actual usefulness of our techniques.

The remainder of this paper is organized as follows. Section 2 shows how to gather changes and their effects; Section 3 applies this to CVS. Section 4 describes the basic approaches to mining these data, followed by examples in Section 5. In Section 6, we 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 7 discusses related work and Section 8 closes with conclusion and consequences.



A) The user inserts a new preference into the field fKeys[]

B) ROSE suggests locations for further changes, e.g. the function initDefaults()

```
public final OverlayPreferenceStore.OverlayKey[] fKeys= new OverlayPreferenceStore.OverlayK
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, OPEN_STRUCTURE_COM
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, SYNCHRONIZE_SCROLL
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, SHOW_PSEUDO_CONFLI
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, INITIALLY_SHOW_ANC
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, SHOW_MORE_INFO),
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, IGNORE_WHITESPACE)
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, PREF_SAVE_ALL_EDIT

    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, NEW_PREFERENCE),

    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.STRING, AbstractTextEditor.
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, AbstractTextEditor
    //new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, USE_SPLINES),
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, USE_SINGLE_LINE),
    //new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, USE_RESOLVE_UI),
};


public static void initDefaults(IPreferenceStore store) {
    store.setDefault(OPEN_STRUCTURE_COMPARE, true);
    store.setDefault(SYNCHRONIZE_SCROLLING, true);
    store.setDefault(SHOW_PSEUDO_CONFLICTS, false);
    store.setDefault(INITIALLY_SHOW_ANCESTOR_PANE, false);
    store.setDefault(SHOW_MORE_INFO, false);
    store.setDefault(IGNORE_WHITESPACE, false);
    store.setDefault(PREF_SAVE_ALL_EDITORS, false);
    //store.setDefault(USE_SPLINES, false);
    store.setDefault(USE_SINGLE_LINE, true);
```

Related Changes

Symbol	File	Support	Confidence
initDefaults(IPreferenceStore store)	ComparePreferencePage.java	8	1.0
org.eclipse.compare/plugin.properties	plugin.properties	7	0.875
org.eclipse.compare/buildnotes_compare.html	buildnotes_compare.html	6	0.75
TextMergeViewer(Composite parent, int style, CompareConfiguration configuration)	TextMergeViewer.java	6	0.75
propertyChange(PropertyChangeEvent event)	TextMergeViewer.java	6	0.75
createGeneralPage(Composite parent)	ComparePreferencePage.java	5	0.625
createTextComparePage(Composite parent)	ComparePreferencePage.java	5	0.625
handleDispose(DisposeEvent event)	TextMergeViewer.java	4	0.5

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

2. Processing Change Data

Figure 2 illustrates the basic data flow through our ROSE tool.[1] The *ROSE server* reads a version archive (far left), groups the changes into *transactions,* mines the transactions for *rules* which describe implications between software entities: "If fKeys[] is changed, then initDefaults() is typically changed, too." When the user changes some entity (say, fKeys[]), the *ROSE client* queries the rule set for applicable rules and makes appropriate suggestions for further changes (say, initDefaults()).

We begin by introducing formal definitions for changes, transactions, and affected entities, generalizing the concepts as found in existing version archives. Adopting the notation from [26], a *change* is a mapping $\delta : \mathcal{P} \to \mathcal{P}$, which, when applied, transforms a product $p \in \mathcal{P}$ into a *changed product*

[1]ROSE stands for *Reengineering Of Software Evolution;* it is a non-Rational tool.

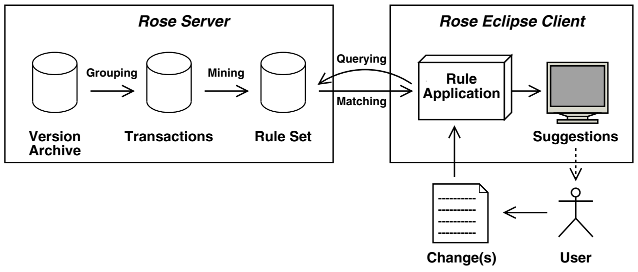


Figure 2. The data flow through ROSE.

$p' = \delta(p) \in \mathcal{P}$. Here, $\mathcal{P}$ is the set of all products; the set of changes is denoted as $\mathcal{C} = \mathcal{P} \to \mathcal{P}$.

Changes can be *composed* using the composition operator $\circ : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$. This is useful for denoting *transactions* consisting of multiple changes to multiple locations. For instance, the transaction $\Delta_{1,2}$ between two versions $p_1, p_2 \in \mathcal{P}$, composed of n individual changes $\delta_1, \ldots, \delta_n$

is expressed as $\Delta_{1,2} = \delta_1 \circ \delta_2 \circ \cdots \circ \delta_n$ with $\Delta_{1,2}(p_1) = (\delta_1 \circ \delta_2 \circ \cdots \circ \delta_n)(p_1) = \delta_1\big(\delta_2\big(\cdots \delta_n(p_1)\big)\big) = p_2$.

To express all the syntactic components affected by a change, we define the concept of *entities.* An entity is a triple (f, c, i), where f is the name of the affected file, c is the syntactic category of the affected component such as *method*, *class*, *file*, ..., and i is the identifier of the affected component. The mapping *entities* retrieves all entities affected by a change or transaction, as in:

$$\mathit{entities}(\Delta) = \mathit{entities}(\delta_1) \cup \cdots \cup \mathit{entities}(\delta_n) =$$
$$\left\{ \begin{array}{lll} (\mathsf{Comp.java}, & \mathit{method}, & \mathit{initDefaults}()), \\ (\mathsf{Comp.java}, & \mathit{field}, & \mathit{fKeys}[]), \\ (\mathsf{Comp.java}, & \mathit{class}, & \mathit{ComparePreferencePage}), \\ (\mathsf{Comp.java}, & \mathit{file}, & \mathsf{Comp.java})^2, \ \ldots \end{array} \right\}$$

Entities are the base for later mining: “I changed one entity; which other entities should I typically change?”

3. Grouping Changes to Transactions

Our ROSE server retrieves changes and transactions as described above from existing version archives—typically from CVS archives, which are frequently used for open-source systems. While CVS is popular, it has some weaknesses that require special *data cleaning* [28]:

Inferring transactions. Most modern version control systems have a concept of *product versioning*—that is, one is able to access transactions as they alter the entire product. CVS, though provides only *file versioning*. To recover per-product transactions from CVS archives, we must *group* the individual per-file changes into individual transactions. ROSE follows the classical *sliding window* approach [7]: two subsequent changes δ_i and δ_{i+1} by the same author and with the same rationale are part of one transaction Δ if they are at most 200 seconds apart.

Branches and merges. The evolution of a product sometimes branches into different evolution strands, which may later be merged again. In a CVS archive, the merge of a branch is not reflected explicitly; instead, the merge becomes a large transaction which includes all the changes made in the branch. In order to detect coupling within transactions, one must take into account all branches, but avoid the large merge transactions. ROSE does so by ignoring all changes that affect more than 30 entities.

Getting entities. CVS has no syntactic knowledge about the files it stores; it manages only files and line numbers for each change. ROSE thus *parses* the files, associating syntactic entities with line ranges. As sketched in Figure 3, ROSE can thus relate any change (given by file and line) to the affected components.

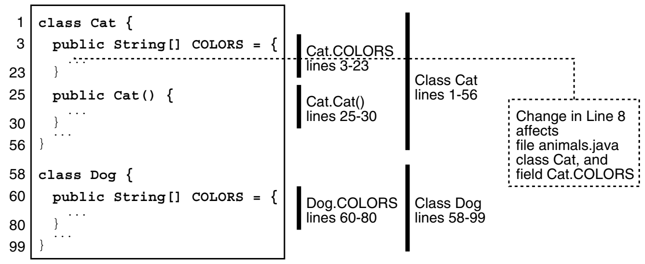

```
1  class Cat {
3    public String[] COLORS = {
       ...
23   }
25   public Cat() {
       ...
30   }
   ...
56 }
58 class Dog {
60   public String[] COLORS = {
       ...
80   }
   ...
99 }
```

Figure 3. Relating changes to entities.

4. From Transactions to Rules

Given the transactions as described in the previous sections, the aim of the ROSE server is to mine *rules* from these transactions. What is a rule? Here is an example:

$$\begin{array}{l} \{(\mathsf{Comp.java}, \mathit{field}, \mathit{fKeys}[])\} \\ \Rightarrow \{\ (\mathsf{Comp.java}, \mathit{method}, \mathit{initDefaults}()), \\ \quad\ (\mathsf{plug.properties}, \mathit{file}, \mathsf{plug.properties})\ \} \end{array} \tag{1}$$

This rule means that whenever the user changes the field *fKeys*[] in Comp.java, then she *should* also change the method *initDefaults*() and the file plug.properties. Here, “should” means that the rule is based on experience and does not constitute absolute truth; the character “$\Rightarrow$” is thus not to be read as a logical implication that always holds.

Formally, an *association rule* r is a pair (x_1, x_2) of two disjoint entity sets x_1 and x_2. In the notation $x_1 \Rightarrow x_2$, x_1 is called the *antecedent* and x_2 the *consequent*.

As said before, rules do not tell an absolute truth. They have a *probabilistic* interpretation based on the *amount of evidence* in the transactions they are derived from. This amount of evidence is determined by two measures:

Support. The support determines 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 plug.properties. Then, the *support* for the above rule is 7.

Confidence. The confidence determines the *strength* of the consequence, or the relative amount of the given consequences across all alternatives. In the above example, the consequence of changing *initDefaults*() and plug.properties applies in 7 out of the 8 transactions involving *fKeys*[]. Hence, the *confidence* for the above rule is $7/8 = 0.875$.

[2] To save space, we abbreviate all file names from Figure 1 to their first syllable; Comp.java stands for ComparePreferencePage.java.

Formally, we define

- the *frequency* of a set x in a set of transactions T as $\mathsf{frq}(T, x) = |\{t \mid t \in T, x \subseteq t\}|$.
- the *support* of a rule $x_1 \Rightarrow x_2$ by a set of transactions T as $\mathsf{supp}(T, x_1 \Rightarrow x_2) = \mathsf{frq}(T, x_1 \cup x_2)$.
- its *confidence* as $\mathsf{conf}(T, x_1 \Rightarrow x_2) = \frac{\mathsf{frq}(T, x_1 \cup x_2)}{\mathsf{frq}(T, x_1)}$.

The shorthand notation $r[s; c]$ denotes a rule r with $s = \mathsf{supp}(T, r)$ and $c = \mathsf{conf}(T, r)$ and a set of transactions T.

4.1. Applying Rules

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, a rule *matches* a set of changed entities if this set is equal to the antecedent.

Assume the programmer has created a sequence of changes $\delta_1 \circ \delta_2 \circ \cdots \circ \delta_k$. The set of changed entities (called *situation*) is $\Sigma = \mathit{entities}(\delta_1 \circ \delta_2 \circ \cdots \circ \delta_k)$. In Figure 1, for instance, the user has extended the variable fKeys[] in file ComparePreferencePage.java. The situation is thus

$$\Sigma = \{(\mathsf{Comp.java}, \mathit{field}, \mathit{fKeys}[])\} \tag{2}$$

How does one compute the suggestions? The set of suggestions for a situation Σ and a set of rules R is defined as the *union* of the consequents of all matching rules:

$$\mathsf{apply}(\Sigma, R) = \bigcup_{(\Sigma \Rightarrow x_2) \in R} x_2$$

In the given situation Σ from (2) and the rule r from (1), ROSE thus suggests the consequent of r:

$$\mathsf{apply}(\Sigma, \{r\}) = \left\{ \begin{array}{l} (\mathsf{Comp.java}, \mathit{method}, \mathit{initDefaults}()), \\ (\mathsf{plug.properties}, \mathit{file}, \mathsf{plug.properties}) \end{array} \right\}$$

The entire set R of actually mined rules contains further rules, though. The actual result of $\mathsf{apply}(\Sigma, R)$ is shown in Figure 1, ordered by confidence.

Let us assume the user decides to follow the first recommendation for initDefaults() (with a confidence of 1.0); it is obvious that a new preference should get a default value. So she changes the method initDefaults(). Again ROSE proposes additional changes, which are in this case the same as before except that now initDefaults() is missing.

Now, the user examines methods createGeneralPage() and createTextComparePage() because they are in the same file as fKeys[] and initDefaults(). Each of these two methods creates a window where preferences can be set. So she extends the createGeneralPage() method, resulting in

$$\Sigma = \left\{ \begin{array}{l} (\mathsf{Comp.java}, \mathit{field}, \mathit{fKeys}[]), \\ (\mathsf{Comp.java}, \mathit{method}, \mathit{initDefaults}()), \\ (\mathsf{Comp.java}, \mathit{method}, \mathit{createGeneralPage}()) \end{array} \right\}$$

Given this situation, a minimum support of 3 and a minimum confidence of 0.5, ROSE computes the following rules:

$$\begin{array}{ll} \Sigma \Rightarrow \{(\mathsf{plug.properties}, \mathit{file}, \mathsf{plug.properties})\} & [5; 1.0] \\ \Sigma \Rightarrow \{(\mathsf{Text.java}, \mathit{method}, \mathit{TextMergeViewer}())\} & [3; 0.6] \\ \Sigma \Rightarrow \{(\mathsf{Text.java}, \mathit{method}, \mathit{propertyChange}())\} & [3; 0.6] \\ \Sigma \Rightarrow \{(\mathsf{build.html}, \mathit{file}, \mathsf{build.html})\} & [3; 0.6] \end{array} \tag{3}$$

Applying the above rules yields the union of the consequents of all rules, because they have the same antecedent. ROSE will rank the entities by their confidence suggesting the user to change the file plug.properties next.

4.2. Computing Rules

ROSE uses the *Apriori Algorithm* [1] to compute association rules. The Apriori Algorithm takes a minimum support and minimum confidence and then computes the set of all association rules in two phases:

1. The algorithm iterates over the set of transactions and forms *entity sets* from the entities that occur in the same transaction. Entity sets that are above the minimal support are called *frequent.* Since an entity set can only be frequent when its subsets are frequent, entity sets are extended in each iteration. This phase finally yields the set F of frequent entity sets.

2. The algorithm computes rules from the sets in F. More precisely, from each of the entity sets $E \in F$ it creates those rules $E - X \Rightarrow X$ where X is a subset of E. (Note that all these rules have the same support $\mathsf{supp}(T, E)$, but different confidences.) Only rules that are above the minimum confidence are returned.

The classical use of the Apriori Algorithm is to compute *all* rules beforehand, and then search the rule set for a given situation. However, computing all rules takes time—several days in our experiments. So we used *two optimizations:*

Constrained antecedents. In our specific case, the antecedent is equal to the situation; hence, we only mine rules *on the fly* which are related to the situation. Mining with such constrained antecedents [24] 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 between two situations.

Single consequents. To speed up the mining process even more, we have modified the approach such that it only computes rules with a single entity in their consequent. So for a situation Σ the rules have the form $\Sigma \Rightarrow \{e\}$. For ROSE, such rules are sufficient because ROSE computes the union of the consequents anyway

(Section 4.1).[3] (Our previous example in (3) already used single consequent rules.)

These optimizations make mining very efficient: The average runtime of a query is about 0.5s for large version histories like GCC.[4]

5. Some Rule Examples

Coupling in GCC. GCC has arrays that define the costs 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:

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

So, whenever the costs type is changed (e.g. for a new operation), ROSE suggests to extend the appropriate cost instances, too.[5]

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:

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

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:

$$\begin{aligned}&\{\ (\text{createuser.sgml}, \textit{file}, \text{createuser.sgml}),\\ &\quad (\text{dropuser.sgml}, \textit{file}, \text{dropuser.sgml})\ \}\\ \Rightarrow &\{\ (\text{createdb.sgml}, \textit{file}, \text{createdb.sgml}), \quad [11; 1.0]\\ &\quad (\text{dropdb.sgml}, \textit{file}, \text{dropdb.sgml})\ \}\end{aligned}$$

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

[3]For each entity $e \in x_2$ in a consequent of a rule $\Sigma \Rightarrow x_2[s; c]$ there exists a single consequent rule $\Sigma \Rightarrow \{e\}[s_e; c_e]$ with higher or equal support and confidence values $s_e \geq s$ and $c_e \geq c$.

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

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

6. Evaluation

After these rule examples, let us now give empirical evidence for the following objectives:

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?

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

Granularity. By default, ROSE suggests changes to *functions* and other fine-grained entities. What are the results if ROSE suggests changes to *files* instead?

6.1. Evaluation Setup

For our evaluation, we analyzed the archives of eight large open-source projects (Table 1 on the next page). 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 Δ whether its entities can be *predicted from earlier history:*

1. We create a *test case* $q = (Q, E)$ consisting of a *query* $Q \subset \textit{entities}(\Delta)$ and an *expected outcome* $E = \textit{entities}(\Delta) - Q$.
2. We take all transactions Δ_i that have been completed before $\textit{time}(\Delta)$ 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 $A_q = \text{apply}(Q, R_{10})$. 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 entities that *matched* the expected outcome and are considered *correct.*
   - $A_q - E_q$ are unexpected recommendations which are *wrong.*

For the assessment of a result A_q, we use two measures from information retrieval [20]: The *precision* P_q describes which fraction of the returned entities was actually expected

Project, Description	History (Training)				Evaluation	
	in CVS since	# Txns	# Txns/Day	# Etys/Txn	Period	# Txns
ECLIPSE, integrated environment	2001-04-28	46,843	56.0	3.17	2003-03-01 to 03-31	2,965
GCC, compiler collection	1997-08-11	47,424	22.4	3.90	2003-04-01 to 04-30	1,083
GIMP, image manipulation tool	1997-01-01	9,796	4.1	4.54	2003-02-01 to 07-31	1,305
JBOSS, application server	2000-04-22	10,843	9.0	3.49	2003-04-01 to 07-31	1,320
JEDIT, text editor	2001-09-02	2,024	2.9	4.54	2003-02-01 to 07-31	577
KOFFICE, office suite	1998-04-18	20,903	11.2	4.25	2003-02-01 to 05-31	1,385
POSTGRESQL, database system	1996-07-09	13,477	5.4	3.27	2003-01-01 to 05-31	925
PYTHON, language + library	1990-08-09	29,588	6.2	2.62	2003-05-01 to 07-31	1,201

Table 1. Analyzed projects (Txn = Transaction; Ety = Entity)

by the user. The *recall* R_q indicates the percentage of expected entities 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 entities are returned (A_q is empty), we define the precision as $P_q = 1$, and in case no entities are expected, we define the recall as $R_q = 1$.

Our goal is to achieve *high precision* and *high recall* values (near 1)—that is to recommend *all* (recall of 1) and *only* expected entities (precision of 1).

For each query q_i, we get a precision-recall pair (P_{q_i}, R_{q_i}). To get an overall measure for the entire history, we summarize these pairs into a single pair using two different averaging techniques from information retrieval:

Macro-evaluation simply takes the mean value of the precision-recall pairs:

$$P_M = \frac{1}{N}\sum_{i=1}^{N} P_{q_i} \qquad R_M = \frac{1}{N}\sum_{i=1}^{N} R_{q_i}$$

This approach uses the precision and recall which have been computed for each query. As users usually think in queries macro-evaluation is sometimes referred to as a *user-oriented* approach—it determines the predictive strength of individual queries.

Micro-evaluation in contrast builds an average precision-recall pair based on entities. It does not use the precision and recall values of single queries, but the sums of returned, matching and expected entities of all queries.

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

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 *summarizing all queries* like "every *n*th suggestion is wrong/correct". For example, the precision P_μ for PYTHON is 0.50: Every second suggestion is correct, which means that the recommended entity was actually changed later on. Micro-evaluation is sometimes referred to as a *system-oriented* approach, because it focuses on the *overall performance* of the system and not on the average query performance.

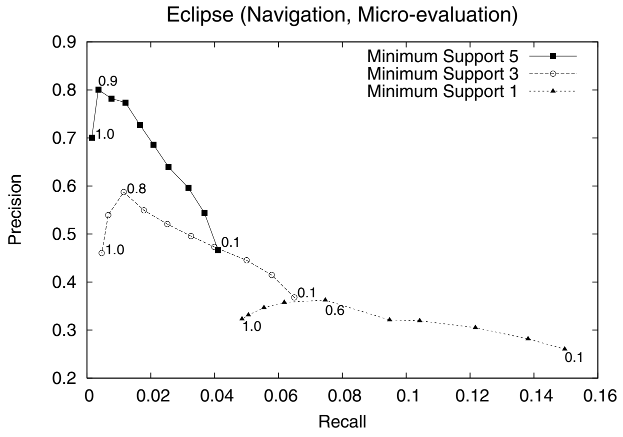


Figure 4. Varying support and confidence

Unless otherwise noted, all averages are given by micro-evaluation.

6.2. Precision vs. Recall

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). We wanted to evaluate the predictive power of ROSE in this situation. For each transaction Δ, and each entity $e \in \mathit{entities}(\Delta)$, we queried $Q = \{e\}$, and checked whether ROSE would predict $E = \mathit{entities}(\Delta) - \{e\}$. For each transaction, we thus tested $|\mathit{entities}(\Delta)|$ queries, each with one element.

Figure 4 shows a so-called *precision-recall graph* with the results for the ECLIPSE project. For each combination of minimum support and minimum confidence the resulting precision-recall pair is plotted. Additionally, subsequent confidence thresholds having the same support are connected with lines. As a result we get three *precision-recall curves*, one for each investigated support. (The connecting lines between measured values are for sake of clarity and not for interpolation.)

In Figure 4, ROSE achieves for a support of 1 and a confidence of 0.1 a recall of 0.15 and a precision of 0.26:

- The *recall* of 0.15 states that ROSE's suggestion correctly included 15% of all changes that were actually carried out in the given transaction.
- The *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 programmer has to check about four suggestions in order to find a correct one.

Figure 4 also shows that *increasing* the support threshold also *increases* the precision, but *decreases* the recall as ROSE gets more cautious. However, using the highest possible thresholds does not always yield the best precision and recall values: If we increase the confidence threshold above 0.80, *both* precision and recall decrease. Furthermore, Figure 4 shows that high support *and* confidence thresholds are required for a high precision. Still, such values result in a very low recall—indicating a trade-off between precision and recall.

In practice, a graph such as the one in Figure 4 is thus necessary to select the "best" support and confidence values for a specific project. In the remainder of this paper, though, we have chosen values that are common across all projects, in order to facilitate comparison.

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

6.3. Likelihood

While a precision like 26% sounds low, keep in mind that this is the likelihood of *each single recommendation* predicting a specific location. If some change in A results in either B, C or D being changed, ROSE suggests B, C, and D, but each suggestion has an average precision of only 33%.

To assess the actual usefulness for the programmer, we 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:

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

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

If, in the example above, ROSE always suggested B, C, and D as topmost suggestions, $L_3 = 100\%$ would hold.

6.4. Results: Navigation through Source Code

We repeated the experiment from Section 6.2 for all eight projects with a support threshold of 1 and a confidence threshold of 0.1—such that for navigation, the user gets several recommendations. The results are shown in Table 2 on the next page (column *Navigation*). For these settings the average recall is 15%, the average precision is 26%; these values are also found for ECLIPSE (Section 6.2). The average likelihood L_3 of the three topmost suggestions predicting a correct location is 64%.

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.

6.5. Results: Error Prevention

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 that have not been changed. If there are, it issues a pop-up window with a warning; it also suggests one or more "missing" entities that should be considered.

We wanted to determine in how many cases ROSE can predict such a missing entity. For this purpose, we took each transaction, left out one entity and checked if ROSE could predict the missing entity. In other words, the query was the complete transaction without the missing entity. So, for each single transaction Δ, and each entity $e \in \mathit{entities}(\Delta)$, we queried $Q = \mathit{entities}(\Delta) - \{e\}$, and checked whether ROSE would predict $E = \{e\}$. For each transaction, we thus again ran $\left|\mathit{entities}(\Delta)\right|$ tests.

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. Still, we wanted to get as many missing entities as possible, resulting in a support threshold of 3. 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 entity.
- The average *precision* is above 50%. This means that every second recommendation of ROSE is correct, or: If a warning occurs, and ROSE recommends further entities, the user on average has to check only one false recommendation before getting to the correct one.

	Navigation			Prevention		Closure	
Support	1			3		3	
Confidence	0.1			0.9		0.9	
Project	R_μ	P_μ	L_3	R_μ	P_μ	R_M	P_M
ECLIPSE	0.15	0.26	0.53	0.02	0.48	1.0	0.979
GCC	0.28	0.39	0.89	0.20	0.81	1.0	0.953
GIMP	0.12	0.25	0.91	0.03	0.71	1.0	0.978
JBOSS	0.16	0.38	0.69	0.01	0.24	1.0	0.981
JEDIT	0.07	0.16	0.52	0.004	0.59	1.0	0.986
KOFFICE	0.08	0.17	0.46	0.003	0.24	1.0	0.990
POSTGRES	0.13	0.23	0.59	0.03	0.66	1.0	0.989
PYTHON	0.14	0.24	0.51	0.01	0.50	1.0	0.986
Average	0.15	0.26	0.64	0.04	0.50	1.0	0.980

Table 2. Results for fine granularity (R = recall; P = precision; L = likelihood)

	Navigation			Prevention		Closure	
Support	1			3		3	
Confidence	0.1			0.9		0.9	
Project	R_μ	P_μ	L_3	R_μ	P_μ	R_M	P_M
ECLIPSE	0.17	0.26	0.54	0.03	0.48	1.0	0.980
GCC	0.44	0.42	0.87	0.29	0.82	1.0	0.946
GIMP	0.27	0.26	0.90	0.08	0.74	1.0	0.963
JBOSS	0.25	0.37	0.64	0.05	0.44	1.0	0.980
JEDIT	0.25	0.22	0.68	0.01	0.44	1.0	0.984
KOFFICE	0.24	0.26	0.67	0.04	0.61	1.0	0.971
POSTGRES	0.23	0.24	0.68	0.05	0.59	1.0	0.978
PYTHON	0.24	0.36	0.60	0.03	0.67	1.0	0.991
Average	0.26	0.30	0.70	0.07	0.66	1.0	0.973

Table 3. Results for coarse granularity (R = recall; P = precision; L = likelihood)

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.

6.6. 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 Δ, we queried $Q = entities(\Delta)$, and checked whether ROSE would predict $E = \emptyset$; we thus had one test per transaction.

As the expected outcome is the empty set, the recall is always 1. To measure the number of false warnings we cannot use micro-evaluation anymore, as one single false alarm results in a summarized precision of 0. We thus turn to *macro-evaluation* precision: The precision for a single query in this setting is either 0 if at least one entity is recommended, or 1 if no entities are recommended; P_M is the percentage of commits where ROSE has not issued a warning, and $1 - P_M$ 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.

6.7. Results: Granularity

By default, ROSE recommends entities at a fine granularity level, e.g. variables or functions. This results in a low coverage of the rules for a project as most functions are rarely changed. Our hypothesis was that if we applied mining to *files* rather than to variables or functions, we would get a higher support (and thus a higher recall).

Therefore, we repeated the experiments from Sections 6.4 to 6.6 with a *coarse granularity*—e.g. mining and applying rules between *files* rather than between entities. The results are shown in Table 3. It turns out that the coarser granularity increases recall in *all* cases (sometimes even dramatically, as the factors 3–8 in KOFFICE show). The precision stays comparable or is increased as well.

If ROSE thus suggests only a file rather than an entity, the suggestions become more frequent and more precise. However, each single suggestion becomes less useful, as it suggests a less specific location—namely only a file rather than a precise entity.[6]

A possible consequence of this result is to have ROSE start with rather vague suggestions (say, regarding files or packages), which become more and more specific as the user progresses. We plan to apply and extend *generalized association rules* [23] such that ROSE can suggest the *finest granularity* wherever possible.

*When given one changed **file,** ROSE can predict 26% of the files actually changed in the same transaction. In 70% of all transactions, ROSE's topmost three suggestions contain a correct location.*

6.8. Threats to Validity

We have studied 10,761 transactions of eight open-source programs. Although the programs themselves are very different, we cannot claim that their version histories would be *representative for all kinds of software projects.* In particular, our evaluation does not allow any conclusions about the predictive power for closed-source projects. However, a stricter software process would result in higher precision and higher recall—and hence, a better predictability.

[6]This is a general trade-off: If all entities were contained within one file, then any suggestion regarding this one file would yield a precision of 100% and a recall of 100%—and be totally useless at the same time.

Transactions do not record the *order* of the individual changes involved. Hence, our evaluation cannot take the order into account the changes were made—and treats all changes equal. In practice, we expect specific orderings of changes to be more frequent than others, which may affect results for navigation and prevention.

We have made no attempt to assess the *quality* of transactions—ROSE learned from past transactions, regardless of whether they may be desired or not. Consequently, the rules learned and evaluated may reflect good practices as well as bad practices. However, we believe that competent programmers make more "good" transactions than "bad" transactions; and thus, there is more good than bad to learn from history.

We have examined the predictive power of ROSE and assumed that suggesting a change, narrowed down to a single file or even a single entity, would be *useful.* However, it may well be that missing related changes could be detected during compilation or tests (in which case ROSE's suggestions would not harm), or may be known by trained programmers anyway (who may find ROSE's suggestions correct, but distracting). Eventually, usefulness for the programmer can only be determined by studies with real users, which we intend to accomplish in the future.

7. Related Work

Independently from us, Annie Ying developed an approach that also uses association rule mining on CVS version archives [25]. 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, though, Ying's tool can only suggest files, not finer-grained entities, and does not support mining-on-the-fly.

Change data has been used by various researchers for quantitative analyses. Word frequency analysis and keyword classification of log messages can identify the purpose of changes and relate it to change size and time between changes [18]. Various researchers computed metrics on the module or file level [3, 9, 11, 12] or orthogonal to these per feature [19] and investigated the change of these metrics over time, i.e. for different releases or versions of a system.

Gall et al. were the first to use release data to detect logical coupling between modules [8]. The CVS history allows to detect more fine-grained logical coupling between classes [10], files and functions [27]. None of these works on logical coupling did address its predictive power. Sayyad-Shirabad et al. use inductive learning to learn different concepts of relevance between logically coupled files [21, 22]. A concept is a set of *attributes* like file name, extension and simple metrics like number of routines defined. If two files have these attributes, then they are relevant to each other. Sayyad-Shirabad thoroughly evaluated the predictive power of the concepts found, but none of the papers gives 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 [16] or generalized association rules [17]. The latter take 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 [5] or program code [2]. HIPIKAT [6] 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.

8. 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: 44% of related files and 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 4–7% 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").

Program analysis. Another yet unused data source is program analysis; although our approach can detect coupling between items that are not even programs, knowing about the semantics of programs could help separating related changes into likely and non-likely. Furthermore, coupling that can be found via analysis [25] need not be repeated in ROSE's suggestions.

Rule presentation. The rules as detected by ROSE describe the factual software process—which may or may not be the intended process. Consequently, these rules can and should be made explicit. In previous work [27], we used visual mining to detect regularities and irregularities of logically coupled items. Such visualizations could further explain the recommendations to programmers and managers.

We are currently making ROSE available as a plug-in for ECLIPSE. For information on download and installation, see

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

Acknowledgments. This project is funded by the Deutsche Forschungsgemeinschaft, grant Ze 509/1-1. Holger Cleve, Carsten Görg, Christian Lindig, Stefan Siersdorfer, and the anonymous ICSE reviewers 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] T. Ball, J.-M. Kim, A. A. Porter, and H. P. Siy. If your version control system could talk.... In *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*, 1997.

[4] J. M. Bieman, A. A. Andrews, and H. J. Yang. Understanding change-proneness in OO software through visualization. In *Proc. 11th International Workshop on Program Comprehension*, pages 44–53, Portland, Oregon, May 2003.

[5] 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 [14], pages 364–374.

[6] D. Čubranić and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In ICSE 2003 [13], pages 408–418.

[7] K. Fogel and M. O'Neill. *cvs2cl.pl: CVS-log-message-to-ChangeLog conversion script*, Sept. 2002. http://www.red-bean.com/cvs2cl/.

[8] 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.

[9] H. Gall, M. Jazayeri, R. Klösch, and G. Trausmuth. Software Evolution Observations based on Product Release History. In *Proceedings of International Conference on Software Maintenance (ICSM '97)*, pages 160–196, 1997.

[10] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In IWPSE 2003 [15], pages 13–23.

[11] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7), 2000.

[12] A. E. Hassan and R. Holt. The chaos of software development. In IWPSE 2003 [15].

[13] *Proc. 25th International Conference on Software Engineering (ICSE)*, Portland, Oregon, May 2003.

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

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

[16] 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.

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

[18] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proc. International Conference on Software Maintenance (ICSM 2000)*, pages 120–130, San Jose, California, USA, Oct. 2000. IEEE.

[19] A. Mockus, D. M. Weiss, and P. Zhang. Understanding and predicting effort in software projects. In ICSE 2003 [13], pages 274–284.

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

[21] J. Sayyad-Shirabad, T. C. Lethbridge, and S. Matwin. Supporting maintainance of legacy software with data mining techniques. In ICSM 2001 [14], pages 22–31.

[22] J. Sayyad-Shirabad, T. C. Lethbridge, and S. Matwin. Mining the maintenance history of a legacy software system. In *Proc. International Conference on Software Maintenance (ICSM 2003)*, Amsterdam, Netherlands, Sept. 2003. IEEE.

[23] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proceedings of the 21th Very Large Data Bases Conference (VLDB)*, pages 407–419, 1995.

[24] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In *Proceedings of the 3rd International Conference on KDD and Data Mining (KDD '97)*, Newport Beach, California, USA, Aug. 1997.

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

[26] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, Feb. 2002.

[27] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In IWPSE 2003 [15], pages 73–83.

[28] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. Technical report, Saarland University, Mar. 2004. Submitted for publication.