Mining Version Histories to Guide Software Changes

Thomas Zimmermann, *Student Member*, *IEEE*, Peter Weißgerber,
Stephan Diehl, and Andreas Zeller, *Member*, *IEEE Computer Society*

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, the mined association rules 1) suggest and predict likely further changes, 2) show up item coupling that is undetectable by program analysis, and 3) can prevent errors due to incomplete changes. After an initial change, our ROSE prototype can correctly predict further locations to be changed; the best predictive power is obtained for changes to existing software. In our evaluation based on the history of eight popular open source projects, ROSE's topmost three suggestions contained a correct location with a likelihood of more than 70 percent.

Index Terms—Programming environments/construction tools, distribution, maintenance, enhancement, configuration management, clustering, classification, association rules, data mining.

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[1] 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**. If a programmer wants to commit changes, but has missed a related change, ROSE issues a warning.
- **Detect coupling undetectable by program analysis**. As ROSE operates exclusively on the version history, it is able to detect coupling between items that cannot be detected by program analysis.

All ROSE needs for this task is a version archive, such as CVS; a simple parser that decomposes files into fine-grained entities such as classes, functions, or sections is optional.

1. ROSE is short for *Reengineering of Software Evolution*.

- *T. Zimmermann and A. Zeller are with the Department of Computer Science, Saarland University, Postfach 15 11 50, 66041 Saarbrücken, Germany. E-mail: zimmerth@cs.uni-sb.de, zeller@acm.org.*
- *P. Weißgerber and S. Diehl are with the Computer Science Department, Catholic University Eichstätt, Ostenstr. 14, 85072 Eichstätt, Germany. E-mail: peter.weissgerber@ku-eichstaett.de, diehl@acm.org.*

Manuscript received 22 Oct. 2004; revised 8 Apr. 2005; accepted 19 Apr. 2005; published online 29 June 2005.
Recommended for acceptance by A. Hassan and R. Holt.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-0221-1004.

ROSE is not the first tool to leverage version histories. In earlier work (Section 8), researchers have used history data to understand programs and their evolution [3], to detect evolutionary coupling between files [10] or classes [5], or to support navigation in the source code [8].

In contrast to this state of the art, the present work

- detects coupling between fine-grained *program entities* such as functions or variables (rather than, say, classes), thus increasing locality and integrating with program analysis;
- distinguishes between different kinds of changes by obtaining *multidimensional* association rules;
- thoroughly evaluates the *ability to predict related or missing changes*, thus evaluating the actual usefulness of our techniques; and
- investigates how the predictive power *changes over the lifetime of a project* and is influenced by activities, releases, and the length of the version history.

The remainder of this paper is organized as follows: Section 2 introduces evolutionary coupling. Section 3 describes the architecture and usage scenarios of our ROSE tool; Section 4 applies this to CVS. Section 5 describes the basic approaches to mining these data, followed by examples in Section 6. In Section 7, 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 8 discusses related work and Section 9 closes with conclusion and consequences.

2 Evolutionary Coupling

The revision history of a software system conveys important information about how and why the system evolved in time. The revision history can also tell us which parts of the system are *coupled* by common changes or

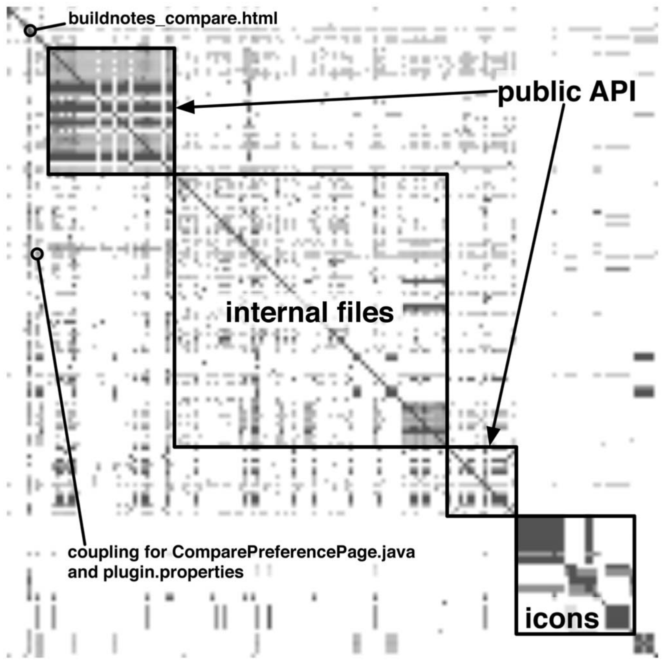
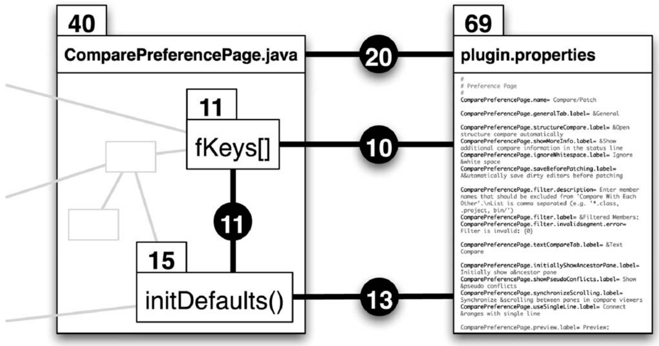


Fig. 1. Common changes of files in an ECLIPSE plug-in.



```
#
# Preference Page
#
ComparePreferencePage.name= Compare/Patch

ComparePreferencePage.generalTab.label= &General

ComparePreferencePage.structureCompare.label= &Open
structure compare automatically
ComparePreferencePage.showMoreInfo.label= &Show
additional compare information in the status line
ComparePreferencePage.ignoreWhitespace.label= Ignore
&white space
ComparePreferencePage.saveBeforePatching.label=
A&utomatically save dirty editors before patching

ComparePreferencePage.filter.description= Enter member
names that should be excluded from 'Compare With Each
Other'.\nList is comma separated (e.g. '*.class,
.project, bin/')
ComparePreferencePage.filter.label= &Filtered Members:
ComparePreferencePage.filter.invalidsegment.error=
Filter is invalid: {0}

ComparePreferencePage.textCompareTab.label= &Text
Compare

ComparePreferencePage.initiallyShowAncestorPane.label=
Initially show a&ncestor pane
ComparePreferencePage.showPseudoConflicts.label= Show
&pseudo conflicts
ComparePreferencePage.synchronizeScrolling.label=
Synchronize &scrolling between panes in compare viewers
ComparePreferencePage.useSingleLine.label= Connect
&ranges with single line

ComparePreferencePage.preview.label= Preview:
```

Fig. 2. Evolutionary coupling in ECLIPSE.

cochanges: "Whenever the database schema was changed, the `sqlquery()` method was altered, too." We call such dependencies *evolutionary coupling* to reflect that they result from evolution—in contrast to *logical coupling,* as determined by program analysis.

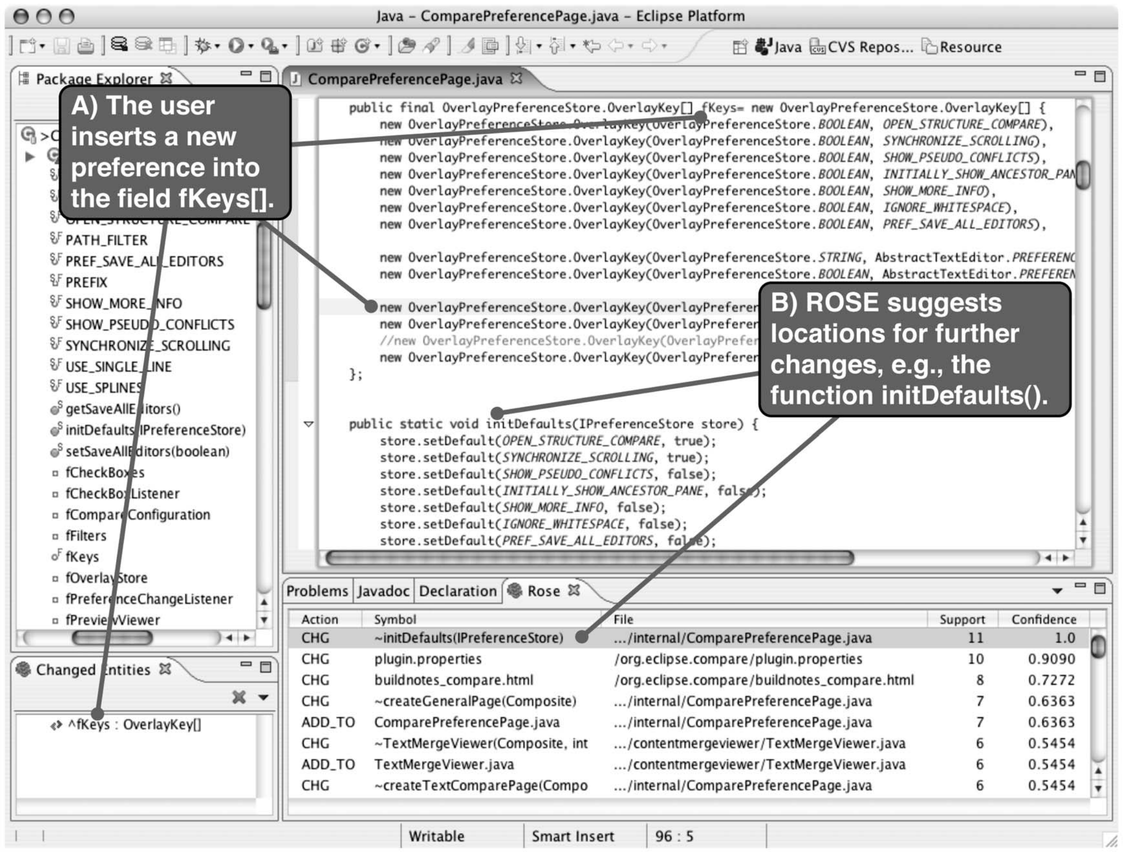
As an example of evolutionary coupling, consider Fig. 1, visualizing evolutionary coupling within the `org.eclipse.compare` plug-in of the ECLIPSE programming environment.[2] Each pixel represents the coupling between two files; the darker the pixel, the stronger the coupling. The files are sorted by the surrounding directory. This highlights directories with a tight evolutionary coupling as blocks along the diagonal, e.g., the public API directory or the icon directories of the plug-in. This coupling is obtained from a version archive such as CVS.

Fig. 2 shows a specific coupling within the `org.eclipse.compare` plug-in—the coupling between the files `ComparePreferencePage.java` and `plugin.properties`. Each file is listed along with the number of changes: `plugin.properties` has been changed 69 times, for instance.

Both files have been changed together 20 times, indicating some evolutionary coupling. This is not a very strong coupling, though, since `ComparePreferencePage.java` has been changed 40 times overall—that is, it has been changed 20 times *without* `plugin.properties` being changed at the same time.

To obtain more details, we can *increase the granularity* from files to *entities* and determine the evolutionary coupling between the individual attributes and functions defined in `ComparePreferencePage.java`. This reveals new couplings—for instance, a coupling between the `fKeys[]` attribute and the `initDefaults()` method as well as a coupling between the `fKeys[]` attribute and the `plugin.properties` file. Both couplings are strong: In 10 out of 11 times that `fKeys[]` has been changed, `plugin.properties` has been changed, too.

2. Such visualizations can be produced with EPOSEE (www.eposoft.org/eposee), our tool for visual data mining [6].



```
public final OverlayPreferenceStore.OverlayKey[] fKeys= new OverlayPreferenceStore.OverlayKey[] {
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, OPEN_STRUCTURE_COMPARE),
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, SYNCHRONIZE_SCROLLING),
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, SHOW_PSEUDO_CONFLICTS),
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, INITIALLY_SHOW_ANCESTOR_PAN
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, SHOW_MORE_INFO),
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, IGNORE_WHITESPACE),
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, PREF_SAVE_ALL_EDITORS),

    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.STRING, AbstractTextEditor.PREFERENC
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, AbstractTextEditor.PREFEREN

    new OverlayPreferenceStore.OverlayKey(OverlayPreferen
    new OverlayPreferenceStore.OverlayKey(OverlayPreferen
    //new OverlayPreferenceStore.OverlayKey(OverlayPrefer
    new OverlayPreferenceStore.OverlayKey(OverlayPreferen
};


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);
```

Action	Symbol	File	Support	Confidence
CHG	~initDefaults(IPreferenceStore)	.../internal/ComparePreferencePage.java	11	1.0
CHG	plugin.properties	/org.eclipse.compare/plugin.properties	10	0.9090
CHG	buildnotes_compare.html	/org.eclipse.compare/buildnotes_compare.html	8	0.7272
CHG	~createGeneralPage(Composite)	.../internal/ComparePreferencePage.java	7	0.6363
ADD_TO	ComparePreferencePage.java	.../internal/ComparePreferencePage.java	7	0.6363
CHG	~TextMergeViewer(Composite, int	.../contentmergeviewer/TextMergeViewer.java	6	0.5454
ADD_TO	TextMergeViewer.java	.../contentmergeviewer/TextMergeViewer.java	6	0.5454
CHG	~createTextComparePage(Compo	.../internal/ComparePreferencePage.java	6	0.5454

Fig. 3. 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.

In the past, researchers have leveraged evolutionary coupling to provide insights about a software system [12], [35]. In this work, we use evolutionary coupling to *guide programmers along related changes.*

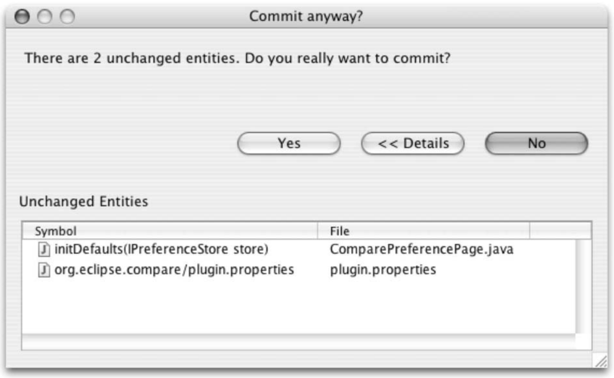
3 ROSE IN A NUTSHELL

Our approach has been realized in the ROSE tool. ROSE learns from the information stored in version archives to make recommendations for programmers. These recommendations are useful in two common scenarios.

- **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 related changes in a view.

  Fig. 3 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 0.727—suggesting that after adding the new preference, the documentation should be updated, too. Such a dependency is undetectable by program analysis.
- **The "Prevent Errors" scenario**. Besides supporting navigation, ROSE should also *prevent errors*. The scenario is that when a user decides to commit changes to the version archive, ROSE checks if there are related changes with a *high confidence* that have not been changed yet. If there are, like the top two locations in Fig. 3, ROSE issues a pop-up window with a warning. It also suggests the "missing" item that should be considered, as in Fig. 4.



There are 2 unchanged entities. Do you really want to commit?

Unchanged Entities

Symbol	File
initDefaults(IPreferenceStore store)	ComparePreferencePage.java
org.eclipse.compare/plugin.properties	plugin.properties

Fig. 4. ROSE points programmers to locations they have likely missed.

In terms of architecture, 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 sped up and rules are always up-to-date.

More details about preprocessing are described in Section 4 and in [36]. The mining phase is explained in Section 5.

4 From Changes to Transactions

In order to describe how ROSE works, we need a few definitions. To represent the syntactic components being changed, we define the concept of *entities*. An entity is a triple (c, i, p), where c is the syntactic category, i is the identifier, and p is the parent entity or $\perp$ for the root entity. For example, the entity

$$(method, initDefaults(),$$
$$(class, Comp, (file, Comp.java, \dots)))$$

represents the method `initDefaults()` of the class `Comp` in the file `Comp.java`. For the syntactic categories, we will use different granularities as discussed in Section 5.4.

We distinguish different kinds of changes using the following predicates, each of them represents a different dimension for the later multidimensional data mining:

- An entity e has been altered: $alter(e)$.
- Some entity has been added into another entity e: $add\_to(e)$.
- Some entity has been deleted from another entity e: $del\_from(e)$.

A *transaction* is the set of changes simultaneously submitted by a developer to a version archive, e.g.,[3]

$$T = \left\{ \begin{array}{llll} alter(& method, & initDefaults(), & \dots), \\ alter(& field, & fKeys[], & \dots), \\ alter(& class, & Comp, & \dots), \\ alter(& file, & Comp.java, & \dots), \\ add\_to(& file, & Comp.java, & \dots), \\ \quad\vdots & & & \end{array} \right\}.$$

The elements of a transaction are called *items*—each of them represents a change—and are the base for later mining: "I *altered* (or *added*) one entity; which other entities should I typically *change*?"

3. To save space, we abbreviate all file and class names from Fig. 3 to their first syllable; for instance, `Comp.java` stands for `ComparePreferencePage.java`.

Our ROSE tool retrieves changes and transactions as described above from existing version archives—typically from CVS [4] archives, which are frequently used for open-source systems. While CVS is popular, it has some weaknesses that require special *data cleaning* [36]:

- **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 [9]: Two subsequent changes by the same author and with the same log message are part of one transaction if they are at most 200 seconds apart. The sliding time window approach is important to capture long transactions that might be split otherwise [36].
- **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 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 for syntactic entities like classes, functions, fields within source code, or sections within documentation. Next, ROSE associates these syntactic entities with line ranges. As sketched in Fig. 5, ROSE can thus relate any change (given by file and line) to the affected components.

5 From Transactions to Rules

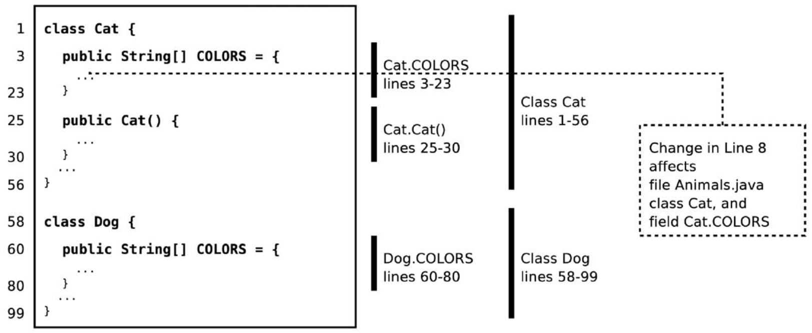
Given the transactions as described in the previous sections, the aim of the ROSE tool is to mine *rules* from these transactions. Here is an example of such a rule:

$$\{alter(field, fKeys[], \dots)\}$$
$$\Rightarrow \left\{ \begin{array}{l} alter(method, initDefaults(), \dots), \\ alter(file, plug.properties, \dots) \end{array} \right\}. \quad (1)$$

This rule means that whenever the user alters the field `fKeys[]`, then she should also alter the method `initDefaults()` and the file `plug.properties`.

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

Rules have a *probabilistic* interpretation based on the *amount of evidence* in the transactions they are derived from. (Hence, the character "$\Rightarrow$ " is to be read as a *possible* rather than an absolute implication.) The amount of evidence is determined by two measures:



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

Fig. 5. Relating changes to entities.

- **Support count**. The support count determines the *number* of transactions the rule has been derived from. Assume that the field `fKeys[]` was altered in 11 transactions. Of these 11 transactions, 10 also included changes of type *alter* of both the method `initDefaults()` and the file `plug.properties`. Therefore, the *support count* for the above rule is 10.
- **Confidence**. The confidence determines the *strength* of the consequence, or the relative amount of the given consequences across all alternatives for a given antecedent. In the above example, the consequence of changing `initDefaults()` and `plug.properties` applies in 10 out of the 11 transactions involving `fKeys[]`. Hence, the *confidence* for the above rule is $10/11 = 0.909$.

Formally, we define

- the (occurrence) *frequency* of a set x in a set of transactions D as $\texttt{freq}_D(x) = |\{t \mid t \in D, x \subseteq t\}|$.
- the *support count* of a rule $x_1 \Rightarrow x_2$ by a set of transactions D as[4]

$$\texttt{count}_D(x_1 \Rightarrow x_2) = \texttt{freq}_D(x_1 \cup x_2).$$

- the *confidence* of a rule $x_1 \Rightarrow x_2$ in a set of transactions D as $\texttt{conf}_D(x_1 \Rightarrow x_2) = \frac{\text{count}_D(x_1 \Rightarrow x_2)}{\text{freq}_D(x_1)}$.

The shorthand notation $r[s;c]_D$ denotes a rule r with $s = \texttt{count}_D(r)$ and $c = \texttt{conf}_D(r)$ and a set of transactions D. We omit the set of transactions D if it is known in the context or irrelevant.

5.1 Applying Rules

Assume the programmer has performed some changes. We call the set of items that represent these changes the *situation* Σ. The user can choose if the situation Σ contains all changes since the last start of ECLIPSE or since the last commit operation to CVS. Furthermore, she can clear the situation manually. The situation is updated every time the user saves her changes. Once she is finished with her task and commits all changes to the version control system, the final situation is added to the set of transactions D.

In Fig. 3, the user has extended the variable `fKeys[]` in the file `ComparePreferencePage.java`. The situation is thus

$$\Sigma = \{alter(field, fKeys[], \ldots)\}. \tag{2}$$

Based on the situation, ROSE suggests possible further changes by *applying* matching rules. In general, a rule *matches* a set of items, e.g., the situation, if this set is equal to the antecedent of the rule.

How does ROSE 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:

$$\texttt{apply}_R(\Sigma) = \bigcup_{(\Sigma \Rightarrow x_2) \in R} x_2. \tag{3}$$

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

$$\texttt{apply}_{\{r\}}(\Sigma) = \left\{ \begin{array}{l} alter(method, initDefaults(), \ldots), \\ alter(file, plug.properties, \ldots) \end{array} \right\}.$$

The entire set R of actually mined rules contains further rules, though. The actual result of $\texttt{apply}_R(\Sigma)$ is shown in Fig. 3, 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 alters 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 the 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

$$\Sigma = \left\{ \begin{array}{l} alter(field, fKeys[], \ldots), \\ alter(method, initDefaults(), \ldots), \\ alter(method, createGeneralPage(), \ldots) \end{array} \right\}.$$

4. In data mining, the *support* $\texttt{supp}_D(x_1 \Rightarrow x_2) = (\texttt{count}_D(x_1 \Rightarrow x_2)/|D|)$ is frequently preferred over the support count. ROSE relies on the *support count* rather than support for two reasons: 1) *Support count is easy to understand for developers*. A support count of 7 clearly indicates the frequency of a rule in the past. In contrast, it is difficult to assess support values like 0.000145 without any additional knowledge (such as the total number of transactions). 2) *Support count can be reused for different projects*. In any project, a support count of 10 represents a rather strong evolutionary coupling. The total number of transactions has only a small impact on this coupling, since more transactions distribute across more entities (see Table 1). In practice, one should normalize both support and support count against a combination of the average transaction size or the average number of changes per item. However, choosing the right normalization is research on its own.

Given this new situation, ROSE computes new rules on the fly as discussed in Section 5.3. Here are the rules that have a minimum support count of 3 and a minimum confidence of 0.4:

$$\begin{array}{ll} \Sigma \Rightarrow \{alter(file, plug.properties, \ldots)\} & [7; 1.00] \\ \Sigma \Rightarrow \{alter(file, build.html, \ldots)\} & [4; 0.57] \\ \Sigma \Rightarrow \{add\_to(file, Comp.java, \ldots)\} & [4; 0.57] \\ \Sigma \Rightarrow \{alter(method, TextMergeViewer(), \ldots)\} & [3; 0.42] \\ \Sigma \Rightarrow \{alter(method, propertyChange(), \ldots)\} & [3; 0.42]. \end{array} \tag{4}$$

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

The number of rules and, thus, the number of recommendations depends on the support count and confidence thresholds that are selected by the user. In general, a user would start with low values like 1 for the support count and 0.1 for the confidence. As a result, she will get all recommendations, but since ROSE ranks by confidence, the strongest ones will be on top of the list.

5.2 Multidimensional Rules

The advantage of *add_to* and *del_from* items is that they abstract from the name of the added or deleted entity to the name of the surrounding entity. Using such items, ROSE learns additional, more general rules.

For instance, in (4), the third rule suggests to add something to the file `Comp.java`. This is due to the fact that preferences are identified with constants in ECLIPSE (see the source code in Fig. 3).

Now, let us assume that the developer started by adding a new constant `FOO` to the file `Comp.java`. In the case of one-dimensional data, the situation will contain a single *change* to `FOO`; thus, ROSE cannot make any recommendations since there is no history for the new constant `FOO`. In contrast, using multidimensional data the situation will contain a single *add_to* item; thus, ROSE uses the surrounding file for mining to make several recommendations:

$$\begin{array}{ll} \{add\_to(file,Comp.java,...)\} \Rightarrow \{alter(file,plug.properties,...)\} & [10;0.83] \\ \{add\_to(file,Comp.java,...)\} \Rightarrow \{alter(file,build.html,...)\} & [10;0.83] \\ \{add\_to(file,Comp.java,...)\} \Rightarrow \{add\_to(file,TextMergeViewer.java,...)\} & [9;0.75] \\ \{add\_to(file,Comp.java,...)\} \Rightarrow \{alter(method,initDefaults(),...)\} & [8;0.66]. \end{array} \tag{5}$$

We reuse the same mining algorithms (see Section 5.3) for both one-dimensional and multidimensional mining. The only difference is the kind of data which is used for mining: For one-dimensional mining, we use *alter* items; for multidimensional mining, we use *alter*, *add_to*, and *del_from* items.

5.3 Computing Rules

The classical approach to compute association rules is the *Apriori Algorithm* [1]. The Apriori Algorithm takes a minimum support count (or minimum support) and a minimum confidence and computes the set of all association rules that are above both thresholds.

The straightforward 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. ROSE uses *two optimizations* to compute rules on demand:

- **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. Mining with such constrained antecedents [31] 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 without recomputing all rules.
- **Single consequents**. To speed up the mining process even more, we have modified the approach such that it only computes rules with a single item 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 (see Section 5.1).[5]

These optimizations make mining very efficient: ROSE needs one database query to find all transactions containing the current situation (constrained antecedents) and another one to compute the confidence values of other items (single consequents). The details of the modified mining algorithm are discussed in [34]. The average runtime of a ROSE query is about 0.5s for large version histories like GCC, measured on a PC with Intel 2.0 GHz Pentium 4 and 1 GB RAM. The parsing time is not included (parsing takes only a fraction of a second, since changed files already reside in memory).

5.4 Granularity

ROSE performs mining on two levels:

- **Fine-granular mining**. For C, C++, JAVA, and PYTHON source code and TEX documentation, we use fine-granular entities, such as fields, functions, and subsections, for the *alter* items. For *add_to* and *del_from* items, we use file entities in any case. All other files like images or text files are represented with an *alter* item for the file entity.
- **Coarse-granular mining**. Regardless of the file type, we only use *alter* items for file entities. Additionally, we use *add_to* and *del_from* items for directory entities to capture when a file has been added or deleted.

Coarse-granular rules have a higher support count and usually return more results. However, they are less precise in the location and, thus, only of limited use for guiding programmers (see Section 7.7).

5. For each item $e \in x_2$ in a consequent of a rule $\Sigma \Rightarrow x_2[s; c]$, there exists a single consequent rule r of the form $\Sigma \Rightarrow \{e\}[s_r; c_r]$ with higher or equal support count and confidence $s_r \geq s$ and $c_r \geq c$ (because $\texttt{freq}(\Sigma \cup \{e\}) \geq \texttt{freq}(\Sigma \cup x_2)$ since $\Sigma \cup \{e\} \subseteq \Sigma \cup x_2$).

6 Some Rule Examples

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

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

$$\{\, alter(type, processor\_cost, (file, i386.h, \ldots))\}$$
$$\Rightarrow \left\{\begin{array}{l} alter(var, i386\_cost, (file, i386.c, \ldots)), \\ alter(var, i486\_cost, (file, i386.c, \ldots)), \\ alter(var, k6\_cost, (file, i386.c, \ldots)), \\ alter(var, pentium\_cost, (file, i386.c, \ldots)), \\ alter(var, pentiumpro\_cost, (file, i386.c, \ldots)) \end{array}\right\}$$
$$[9; 0.82].$$

  So, whenever the costs type is altered (e.g., for a new operation), ROSE suggests to extend the appropriate cost instances, too. This rule also holds for the other direction, with the same support count and (incidentally) the same confidence.
- **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:

$$\{\, alter(func, GrafObj\_getattr(), (file, \_Qdmodule.c, \ldots))\}$$
$$\Rightarrow \{alter(func, outputGetattrHook(), (file, qdsupport.py, \ldots))\}$$
$$[10; 0.91].$$

  Whenever the C function `GrafObj_getattr()` in file `_Qdmodule.c` was altered, so was the PYTHON function `outputGetattrHook()` in file `qdsupport.py`—a classical coupling between interface and implementation. It would require cross-language program analysis to detect this coupling.
- **POSTGRES documentation**. Data mining can reveal coupling between items that are not even programs, as in the POSTGRES documentation for its command line tools:

$$\left\{\begin{array}{c} alter(file, createuser.sgml, \ldots), \\ alter(file, dropuser.sgml, \ldots) \end{array}\right\}$$
$$\Rightarrow \left\{\begin{array}{c} alter(file, createdb.sgml, \ldots), \\ alter(file, dropdb.sgml, \ldots) \end{array}\right\}$$
$$[11; 1.0].$$

  Whenever both `createuser.sgml` and `dropuser.sgml` have been altered, the files `createdb.sgml` and `dropdb.sgml` have been altered, too. The coupling between these files is caused by common options of the documented tools and by examples that have been duplicated.

7 Evaluation

After these rule examples, let us now give empirical evidence for the scenarios we described in Section 3.

- **Navigation through source code**. Given a single change, 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 has made all necessary changes. How often does ROSE erroneously suggest that a change is missing in the error prevention scenario?

We evaluated additional questions that are not directly related to the usage scenarios:

- **Granularity**. By default, ROSE suggests changes to *functions* and other fine-grained entities. What are the results if ROSE suggests only changes to *files* instead?
- **Maintenance**. Additions and deletions are difficult to predict. How well does ROSE perform if it is applied to *maintenance tasks* which do not add new functionality to programs? As a heuristic, we classify transactions that change only existing entities as maintenance.
- **Multiple Dimensions**. ROSE tries to predict *additions and deletions* with special *add_to* and *del_from* items. What is the actual benefit of these additional items?
- **History**. How much of the version history does ROSE need at least to produce useful recommendations? Does the usefulness decrease over time? Furthermore, does the quality of recommendations change with development phases and releases?
- **Recent Changes**. As a system evolves, older changes may become less and less relevant for determining related changes. Would focusing on *recent changes* improve the quality of recommendations?

7.1 Evaluation Setup

For our evaluation, we analyzed the archives of eight large open-source projects (Table 1) chosen to cover a wide range of applications, architectures, and languages. For each archive, we chose at least one full month as our *evaluation period* (Table 2). In this period, we checked for each transaction T whether its items can be *predicted from earlier history:*

1. We created a number of *queries* for a transaction. A query $q = (Q, E)$ consists of a *situation* $Q \subset T$ and an *expected outcome* $E = T - Q$.
   - In the "navigation" scenario, there are $|T|$ queries per transaction T, whose situations each consist of a single item $e \in T$.
   - In the "prevention" scenario, there are $|T|$ queries per transaction T, whose situations each

TABLE 1
History of Analyzed Projects (Txn = Coarse-Grained Transaction, "±" = Standard Deviation)

	Before data cleaning				After data cleaning	
Project, Description	in CVS since	# Files	# Items	# Txns	# Txns	# Items/Txn
ECLIPSE, integrated environment	2001-04-28	34,186	301,199	68,866	53,643	3.18±4.26
GCC, compiler collection	1997-08-11	23,467	160,072	46,564	41,596	3.51±4.01
GIMP, image manipulation tool	1997-01-01	3,834	52,122	12,560	11,393	5.37±5.89
JBOSS, application server	2000-04-22	4,730	47,178	8,877	6,988	3.87±5.14
JEDIT, text editor	2001-09-02	954	9,442	1,244	1,108	8.31±7.01
KOFFICE, office suite	1998-04-18	8,098	81,201	25,195	22,503	4.18±5.02
POSTGRES, database system	1996-07-09	2,684	28,346	14,822	13,022	3.34±4.64
PYTHON, language + library	1990-08-09	3,459	70,372	28,494	22,954	2.72±3.56

consist of the transaction with one removed item e—that is, $T - \{e\}$.
- In the "closure" scenario, there is one query per transaction, whose situation each consist of the full transaction T.

2. For each query $q = (Q, E)$, 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 with respect to Q. This means R contains only constrained rules $Q \Rightarrow \{x\}$ (see Section 5.3).
3. We assume that the user does not work through endless lists of suggestions. Thus, we consider only the *top 10 single-consequent rules* $R_{10} \subset R$ ranked by confidence. In our evaluation, we apply R_{10} to get the result of the query $A_q = \mathtt{apply}_{R_{10}}(Q)$. So, the size of A_q is always less or equal than 10.
4. The result A_q of a query $q = (Q, E)$ consists of two parts:
   - $A_q \cap E$ are the items that *matched* the expected outcome and, therefore, are considered *correct.*
   - $A_q - E$ are unexpected recommendations which are *wrong*.

7.2 Precision, Recall, Likelihood, and Feedback

For the assessment of a result A_q for a query $q = (Q, E)$, we use two measures from information retrieval [25]: The *precision* P_q describes which fraction of the returned items was actually expected. The *recall* R_q indicates the percentage of expected items that were returned.

$$P_q = \frac{|A_q \cap E|}{|A_q|}, \quad R_q = \frac{|A_q \cap E|}{|E|}. \tag{6}$$

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 $R_q = 1$.

Our goal is to achieve *high precision* and *high recall* values (near 1)—that is, to recommend *all* (recall of 1) and *only* expected items (precision of 1). Keep in mind that, if the expected outcome has more than 10 items, the recall can never be 1 because even though all answers may be correct, we only consider the top 10 results.

For each query q, we get a precision-recall pair (P_q, R_q). To get an overall measure for all evaluated queries Z which are generated from all the transactions in the evaluation period, we summarize these pairs into a single pair using the *macroevaluation* averaging technique from information retrieval. Macroevaluation simply takes the mean value of the precision-recall pairs for the queries Z:

$$P_M^{all} = \frac{1}{|Z|}\sum_{q\in Z} P_q \ , \quad R_M^{all} = \frac{1}{|Z|}\sum_{q\in Z} R_q \ . \tag{7}$$

This approach uses the precision and recall which have been computed for each query. As macroevaluation determines the predictive strength for queries, it is sometimes referred to as a *user-oriented* approach.

If ROSE does not return any recommendations for a query q (that is, $A_q = \emptyset$), the precision is 1. Taking such queries into account distorts the average precision. Thus, unless otherwise noted, we consider only the queries Z^* where A_q is not empty:[6]

$$Z^* = \{q \mid q = (Q, E) \in Z, \mathtt{apply}_{R_1}(Q) \neq \emptyset\}, \tag{8}$$

$$P_M = \frac{1}{|Z^*|}\sum_{q\in Z^*} P_q \ , \quad R_M = \frac{1}{|Z^*|}\sum_{q\in Z^*} R_q \ . \tag{9}$$

TABLE 2
Evaluation Periods (Txn = Coarse-Grained Transaction)

Project	Evaluation period	# Txns
ECLIPSE	2003-03-01 to 03-31	2,790
GCC	2003-04-01 to 04-30	604
GIMP	2003-02-01 to 07-31	1,204
JBOSS	2003-04-01 to 07-31	358
JEDIT	2003-02-01 to 07-31	326
KOFFICE	2003-02-01 to 05-31	1,263
POSTGRES	2003-01-01 to 05-31	696
PYTHON	2003-05-01 to 07-31	1,021

6. Z^* is independent from k because $\mathtt{apply}_{R_k}(Q)$ is nonempty if and only if $\mathtt{apply}_{R_1}(Q)$ is nonempty (as $\mathtt{apply}_{R_1}(Q) \subseteq \mathtt{apply}_{R_2}(Q) \subseteq \ldots$ holds).

TABLE 3
Evaluation for Fine Granularity (Txn = Transaction, "±" = Standard Deviation)

	Navigation / Prevention ($\|T\| \geq 2$)			Closure (T of any size)		
Project	# Txns[7]	# Items/Txn	# Queries	# Txns[7]	# Items/Txn	# Queries
ECLIPSE	982	4.97±4.77	4,876	2,443	2.59±3.59	2,443
GCC	473	4.17±3.51	1,972	598	3.51±3.38	598
GIMP	1,090	6.18±6.12	6,733	1,201	5.70±6.02	1,201
JBOSS	148	5.67±5.51	839	340	3.03±4.31	340
JEDIT	288	10.21±7.41	2,941	323	9.21±7.56	323
KOFFICE	700	6.76±5.87	4,729	1,210	4.33±5.29	1,210
POSTGRES	369	6.33±5.98	2,337	659	3.99±5.20	659
PYTHON	402	4.22±3.73	1,696	904	2.43±2.96	904

7. We could not use all transactions from Table 2 for fine granularity because some transactions have been empty. For instance, a line that is inserted between two functions is a coarse-grained change for the surrounding file, but not a fine-grained change for the functions.

Additionally, we measure the percentage of queries where ROSE makes at least one recommendation. We refer to this percentage as the *feedback* $Fb = |Z^*|/|Z|$.

To assess the actual usefulness for the programmer, we also check the *likelihood* whether an 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_k is the likelihood that at least one of the top k recommendations made by ROSE for a query $q = (Q, E)$ is correct:

$$L_k = \frac{\left|\{q \mid q = (Q,E) \in Z, \texttt{apply}_{R_k}(Q) \cap E \neq \emptyset\}\right|}{\left|\{q \mid q = (Q,E) \in Z, \texttt{apply}_{R_k}(Q) \neq \emptyset\}\right|}. \quad (10)$$

If some change in A results in either B_1, B_2, or B_3 being changed, ROSE always suggests B_1, B_2, and B_3, the precision is only 33 percent. Still, the recommendations are useful for the programmer, thus, $L_3 = 100$ percent would hold.

7.3 Precision versus Feedback

A major application for ROSE is the "navigation" scenario: The user changes some entity and ROSE automatically recommends possible future changes in a view (Fig. 3). We evaluated the predictive power of ROSE in this situation.

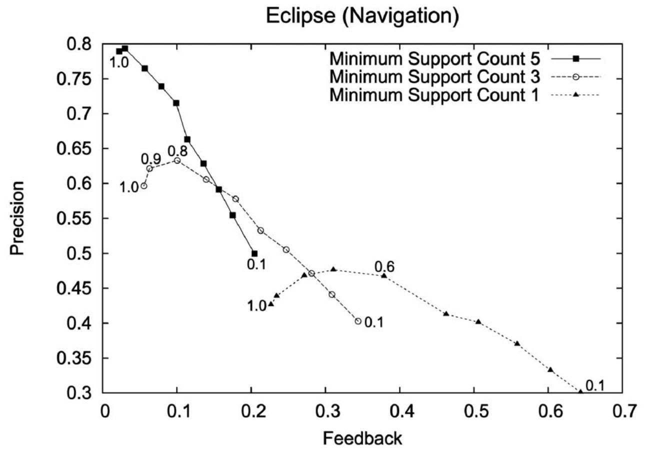


Fig. 6. Varying support count and confidence.

For each transaction T with $|T| \geq 2$ and each item $e \in T$, we considered the situation $Q = \{e\}$ and checked whether ROSE would predict $E = T - \{e\}$. For each transaction, we thus tested $|T|$ situations, each with one element. Table 3 (column *Navigation*) breaks down the evaluated transactions and queries.

Fig. 6 plots the precision against the feedback with the results for the ECLIPSE project. For each combination of minimum support count and minimum confidence, the resulting precision-feedback pair (P_M, Fb) is plotted. For the plot, we prefer the feedback Fb over the recall R_M (which would consider only the set of queries Z^* for which ROSE made suggestions) to take into account that ROSE gets more cautious for higher thresholds. Additionally, values for subsequent confidence thresholds having the same support count are connected with lines. As a result, we get three *precision-feedback curves*, one for each investigated support count. The connecting lines between measured values are for the sake of clarity and not for interpolation.

In Fig. 6, ROSE achieves for a support count of 1 and a confidence of 0.1 a feedback of 0.64 and a precision of 0.30:

- The *feedback Fb* of 0.64 means that, on average, ROSE made, in two out of three queries, at least one suggestion.
- The *recall* R_M of 0.34 (see Table 4, column *Navigation*) states that ROSE's suggestions on average included 34 percent of all changes that were actually carried out in the given transaction.
- The *precision* P_M of 0.30 means that if ROSE made recommendations, on average, 30 percent of them were correct—almost every third suggested change was actually carried out (and, thus, predicted correctly by ROSE). The programmer has to check about three suggestions in order to find a correct one.

Fig. 6 also shows that *increasing* the support count threshold also *increases* the precision, but *decreases* the feedback as ROSE gets more cautious. However, using the highest possible thresholds does not always yield the best precision and feedback values: If we increase for a

TABLE 4
Results for Fine Granularity (R = recall; P = precision; Fb = feedback; L = likelihood)

	Navigation				Prevention			Closure	
Support Count	1				3			3	
Confidence	0.1				0.9			0.9	
Project	*Fb*	R_M	P_M	L_3	*Fb*	R_M	P_M	*Fb*	$1 - Fb$
ECLIPSE	0.64	0.34	0.30	0.57	0.03	0.83	0.70	0.019	0.981
GCC	0.63	0.45	0.31	0.91	0.08	0.96	0.95	0.015	0.985
GIMP	0.60	0.35	0.30	0.92	0.03	0.92	0.89	0.018	0.982
JBOSS	0.59	0.36	0.31	0.62	0.02	0.73	0.65	0.021	0.979
JEDIT	0.74	0.21	0.31	0.86	0.01	0.42	0.38	0.043	0.957
KOFFICE	0.65	0.24	0.23	0.54	0.01	0.50	0.46	0.008	0.992
POSTGRES	0.76	0.29	0.29	0.65	0.02	0.89	0.82	0.012	0.988
PYTHON	0.66	0.37	0.27	0.54	0.02	0.72	0.67	0.013	0.987
Average	0.66	0.33	0.29	0.70	0.03	0.75	0.69	0.019	0.981

support count threshold of three the confidence threshold above 0.80, *both* precision and feedback decrease. The decrease in precision is caused by rules that had a high confidence during the training, which lowered during the evaluation. Furthermore, Fig. 6 shows that high support count *and* confidence thresholds are required for a high precision. Still, such values result in a very low feedback—indicating a trade-off between precise and numerous suggestions.

In practice, a graph such as the one in Fig. 6 is thus necessary to select the "best" support count 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.

Summary. *A well-known trade-off: One can either have* precise *suggestions or* many *suggestions, but not both.*

7.4 Results: Navigation through Source Code

We repeated the experiment from Section 7.3 for all eight projects with a support count 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 4 (column *Navigation*). For these settings, in 66 percent of all queries, ROSE makes recommendations for which the average recall is 33 percent and the average precision is 29 percent. The average likelihood L_3 of the three topmost suggestions predicting a correct location is 70 percent.

While KOFFICE has lower recall, precision, and likelihood values, GCC strikes by overall high values. The reason is that KOFFICE is a project 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.

Summary. *When given one initial item, ROSE makes predictions in 66 percent of all queries. On average, the predictions of ROSE contain 33 percent of all items changed later in the same transaction. For those queries for which ROSE makes recommendations, in 70 percent of the cases, a correct location is within ROSE's topmost three suggestions.*

7.5 Results: Error Prevention

Besides supporting navigation, ROSE should also *prevent errors.* We determined in how many cases ROSE can predict such a missing item. For this purpose, we took each transaction, left out one item, and checked if ROSE could predict the missing item. In other words, the situation was the complete transaction without the missing item. So, for each single transaction T with $|T| \geq 2$ and each item $e \in T$, we considered the situation $Q = T - \{e\}$ and checked whether ROSE would predict $E = \{e\}$. For each transaction, we thus again ran $|T|$ tests. Table 3 (column *Prevention*) breaks down the evaluated transactions and queries.

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 items as possible, resulting in a support count threshold of 3. The results are shown in Table 4 (column *Prevention*):

- The feedback is 3 percent and the average *recall* is about 75 percent. This means that for only one out of every 33 missing items (in GCC: every 13 items), ROSE issues a warning; the percentage of missed alarms is on average 97 percent. However, for those cases where ROSE issues a warning, it predicts 75 percent of the items that are actually missing.
- The average *precision* is above 66 percent. This means that, on average, two out of three recommendations of ROSE are correct, or: If a warning occurs and ROSE recommends further items, the user has to check at most one false recommendation, on average, before getting to the correct one.

Summary. *In 3 percent of the queries where one item is missing, ROSE issues a correct warning. An issued warning predicts on average 75 percent of the items that need to be considered.*

TABLE 5
Evaluation for Coarse Granularity (Txn = Transaction, "±" = Standard Deviation)

	Navigation / Prevention ($\|T\| \geq 2$)			Closure (T of any size)		
Project	# Txns	# Items/Txn	# Queries	# Txns	# Items/Txn	# Queries
ECLIPSE	1,016	4.60±4.75	4,676	2,790	2.31±3.35	2,790
GCC	557	3.33±2.51	1,853	604	3.15±2.49	604
GIMP	1,104	4.26±3.80	4,708	1,204	3.99±3.75	1,204
JBOSS	82	3.38±2.30	277	358	1.54±1.48	358
JEDIT	285	6.44±4.19	1,834	326	5.75±4.31	326
KOFFICE	620	4.60±3.57	2,855	1,263	2.77±3.08	1,263
POSTGRES	320	5.38±4.50	1,722	696	3.01±3.75	696
PYTHON	234	3.25±3.22	760	1,021	1.52±1.80	1,021

7.6 Results: Closure

ROSE is supposed to prevent errors and not to annoy the user issuing false alarms. Thus, for each *complete transaction* T of any size, we considered the situation $Q = T$ and checked whether ROSE would predict $E = \emptyset$; we thus had one test per transaction. Table 3 (column *Closure*) breaks down the evaluated transactions and queries.

As the expected outcome is the empty set, the recall is always 1. Since P_M would summarize only queries that caused a false alarm, the precision would be 0. Thus, we use the feedback Fb to measure the percentage of queries where ROSE has issued a warning, i.e., the percentage of false alarms. $1 - Fb$ is the percentage of queries where ROSE acted correctly.

The results are shown in Table 4 (column *Closure*). One can see that $1 - Fb$ is very high for all projects, usually around 0.98. This means that ROSE issues a false alarm in only every 50th transaction.

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

7.7 Results: Granularity

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

Therefore, we repeated the experiments from Sections 7.4 to 7.6 with a *coarse granularity* (see Section 5.4). Table 5 contains the breakdown of the evaluated transactions to queries; the results are shown in Table 6. It turns out that, for the navigation scenario, the coarse granularity increases recall and feedback in *all* cases (sometimes even dramatically, as a factor of almost 2 for the recalls of JEDIT and KOFFICE shows). For nearly all projects (except JBOSS), the precision stays at the same level or is increased as well. For the error prevention scenario, we observed increases in both recall and precision for some projects (e.g., for PYTHON), as well as drops for others (e.g., for POSTGRES).

TABLE 6
Results for Coarse Granularity (R = recall; P = precision; Fb = feedback; L = likelihood)

	Navigation				Prevention			Closure	
Support Count	1				3			3	
Confidence	0.1				0.9			0.9	
Project	Fb	R_M	P_M	L_3	Fb	R_M	P_M	Fb	$1 - Fb$
ECLIPSE	0.80	0.36	0.29	0.57	0.04	0.83	0.68	0.019	0.981
GCC	0.76	0.59	0.35	0.88	0.21	0.97	0.95	0.040	0.960
GIMP	0.77	0.48	0.28	0.92	0.10	0.94	0.88	0.045	0.955
JBOSS	0.74	0.36	0.19	0.51	0.03	0.56	0.50	0.017	0.983
JEDIT	0.95	0.41	0.31	0.88	0.03	0.41	0.37	0.064	0.936
KOFFICE	0.87	0.45	0.30	0.70	0.04	0.77	0.76	0.021	0.979
POSTGRES	0.95	0.37	0.29	0.72	0.05	0.72	0.63	0.026	0.974
PYTHON	0.73	0.46	0.34	0.61	0.04	0.87	0.82	0.005	0.995
Average	0.82	0.44	0.29	0.72	0.07	0.76	0.70	0.030	0.970

TABLE 7
Evaluation for Maintenance (Txn = Transaction, "±" = Standard Deviation)

Navigation ($\|T\| \geq 2$)	Non-Maintenance			Maintenance		
Project	# Txns	# Items/Txn	# Queries	# Txns	# Items/Txn	# Queries
ECLIPSE	383	5.64±4.77	2,161	599	4.53±4.72	2,715
GCC	159	6.43±4.23	1,022	314	3.03±2.38	950
GIMP	344	11.58±7.15	3,983	746	3.69±3.40	2,750
JBOSS	80	7.24±6.30	579	68	3.82±3.66	260
JEDIT	185	12.64±7.52	2,339	103	5.84±4.75	602
KOFFICE	355	8.55±6.34	3,037	345	4.90±4.68	1,692
POSTGRES	157	9.64±6.92	1,514	212	3.88±3.56	823
PYTHON	183	5.20±4.40	952	219	3.40±2.82	744

If ROSE thus suggests coarse-grained entities, the suggestions become more frequent. However, each single suggestion becomes less useful, as it suggests a less specific location—namely, only a file rather than a precise entity. 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 percent and a recall of 100 percent—and be totally useless at the same time. At the coarse-grained level add_to and del_from, suggestions are useful only if they refer to a directory with specific content (such as `icons/` or `tests/`).

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* [30] such that ROSE can suggest the *finest granularity* wherever possible.

Summary. *When given one altered, added, or deleted* file, *ROSE can suggest further locations in 82 percent of all queries; these recommendations predict, on average, 25 percent of the locations actually modified in the same transaction. For those queries for which ROSE makes recommendations, in 72 percent of the cases, a correct location is within ROSE's topmost three suggestions.*

7.8 Results: Maintenance

We investigated whether ROSE performs better for transactions that consist solely of $alter$ items. We refer to such transactions as *maintenance* transactions. In contrast, *non-maintenance* transactions contain at least one add_to or del_from item. Table 7 breaks down the transactions into maintenance and nonmaintenance.

We repeated the experiments from Section 7.4 for both groups. Table 8 shows the results for the *navigation* scenario. It turns out that, for maintenance tasks, the recall and feedback are higher in *all* cases (sometimes even doubled, as the recalls for ECLIPSE, GCC, or GIMP show). The precision stays virtually unchanged in most cases while the likelihood that the first ten suggestions contain a correct entity increases for some projects (e.g., for ECLIPSE by 8 percent), but decreases for others (e.g., for GCC by 7 percent, for GIMP by 10 percent, and for JEDIT by 9 percent).

Summary. *ROSE has its best predictive power for changes to existing entities. The average recall almost doubles to 44 percent while the precision remains roughly unchanged.*

TABLE 8
Results for Maintenance (R = recall; P = precision; Fb = feedback; L = likelihood)

	Navigation: Support Count=1, Confidence=0.1, Granularity=Fine											
	All				Non-maintenance				Maintenance			
Project	Fb	R_M	P_M	L_{10}	Fb	R_M	P_M	L_{10}	Fb	R_M	P_M	L_{10}
ECLIPSE	0.64	0.34	0.30	0.70	0.55	0.23	0.23	0.66	0.72	0.41	0.36	0.74
GCC	0.63	0.45	0.31	0.91	0.63	0.31	0.35	0.94	0.63	0.60	0.29	0.87
GIMP	0.60	0.35	0.30	0.95	0.58	0.23	0.32	0.99	0.64	0.52	0.28	0.89
JBOSS	0.59	0.36	0.31	0.76	0.55	0.28	0.31	0.76	0.67	0.51	0.37	0.79
JEDIT	0.74	0.21	0.31	0.91	0.73	0.18	0.32	0.93	0.80	0.30	0.29	0.84
KOFFICE	0.65	0.24	0.23	0.69	0.62	0.21	0.23	0.62	0.71	0.29	0.24	0.64
POSTGRES	0.76	0.29	0.29	0.75	0.72	0.22	0.30	0.72	0.83	0.42	0.27	0.71
PYTHON	0.66	0.37	0.27	0.68	0.65	0.30	0.26	0.65	0.67	0.47	0.30	0.68
Average	0.66	0.33	0.29	0.79	0.63	0.25	0.29	0.78	0.71	0.44	0.30	0.77

TABLE 9
Results for Multidimensional Data Mining
(R = recall; P = precision)

	Non-Maintenance: Granularity=Fine, Support Count=1, Confidence=0.1			
	Only changes		All dimensions	
Project	R_M	P_M	R_M	P_M
ECLIPSE	0.15	0.21	0.23	0.23
GCC	0.27	0.35	0.31	0.35
GIMP	0.20	0.31	0.23	0.32
JBOSS	0.23	0.31	0.28	0.31
JEDIT	0.16	0.31	0.18	0.32
KOFFICE	0.14	0.20	0.21	0.23
POSTGRES	0.18	0.29	0.22	0.30
PYTHON	0.20	0.22	0.30	0.26
Average	0.19	0.28	0.25	0.29

7.9 Results: Multiple Dimensions

By default, ROSE considers *alter*, *add_to*, and *del_from* items for its recommendations (Section 5.2). To measure the benefit of the additional dimensions, we repeated the experiments for nonmaintenance from Section 7.8 with a modified version of ROSE that only recommends *alter* items.

The results are shown in Table 9. Recommending all dimensions increases the recall for all projects. The precision improves slightly or remains unchanged.

Summary. *Considering additional dimensions, like additions and deletions, improves the predictive power of ROSE for nonmaintenance tasks. The average recall increases from 19 percent to 25 percent while the precision remains roughly unchanged.*

7.10 Results: History

The previous experiments concerned the predictive power of ROSE based on a fixed evaluation period for each project. In addition, we investigated how the predictive power changes over time: How long does it take until ROSE makes

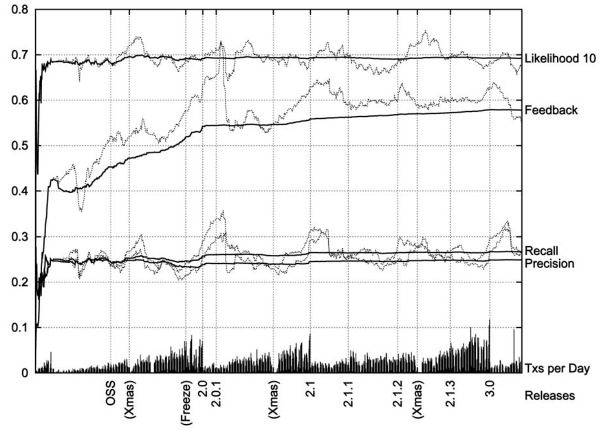


Fig. 7. Predictive power for ECLIPSE related to time (2001-04-28 to 2004-09-14) and releases.

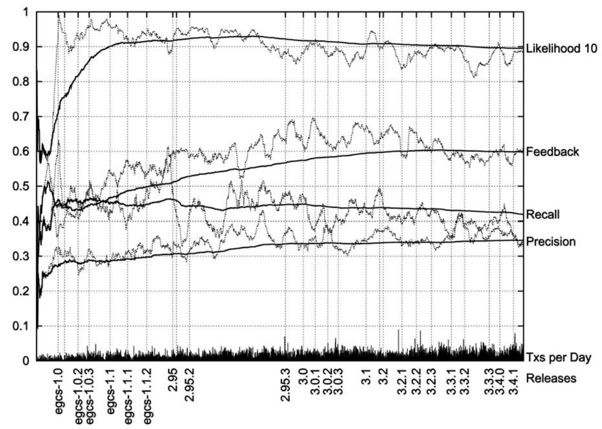


Fig. 8. Predictive power for GCC related to time (1997-08-11 to 2004-08-23) and releases.

useful suggestions and, as the project evolves, does the quality of the suggestions degrade at some point? To answer these questions, we repeated our experiments for the *navigation* scenario for the projects ECLIPSE and GCC using the full history as evaluation period.

For each day, we computed the recall, precision, feedback, and the likelihood that the top 10 suggestions contain at least one correct item, based on *all* transactions before this day, as well as the *moving average* of these measures based on the transactions of the last 42 days. We chose 42 days because, in the ECLIPSE project, this is the duration of the development cycle for one release. The results are illustrated in Fig. 7 for ECLIPSE and in Fig. 8 for GCC, respectively.

Startup. For both ECLIPSE and GCC, the values for each measure increase very quickly. For ECLIPSE, it takes less than a month until recall and precision get close to their maximal values, and even feedback reaches 40 percent after the first month.

Saturation. ROSE learns from all transactions since the beginning of the version history. For GCC, the moving averages indicate that the actual predictive power of ROSE saturates around release 2.95.3 and even decreases after release 3.0 (Fig. 8). The reason for this is that ROSE has much outdated knowledge—which suggests that ROSE should not learn from too old transactions. This decrease is not (yet?) visible for ECLIPSE (Fig. 7).

Peaks. Fig. 7 shows peaks of the moving averages shortly after major releases like 2.0, 2.1, and 3.0 of ECLIPSE. As the moving average lags $42/2 = 21$ days behind the trend, the peaks actually correlate with the release dates. Similar peaks are visible for GCC in Fig. 8, however, not as clearly as for ECLIPSE.

A possible explanation of this phenomenon is that shortly before releases, features are frozen and only existing parts are changed, so we have mostly maintenance transactions (see Section 7.8). As a consequence, the predictive power of ROSE increases at this point. After the actual release, new features are introduced which temporarily decrease the predictive power of ROSE.

Summary. *ROSE learns quickly: A few weeks after a project starts, ROSE makes already useful suggestions.*

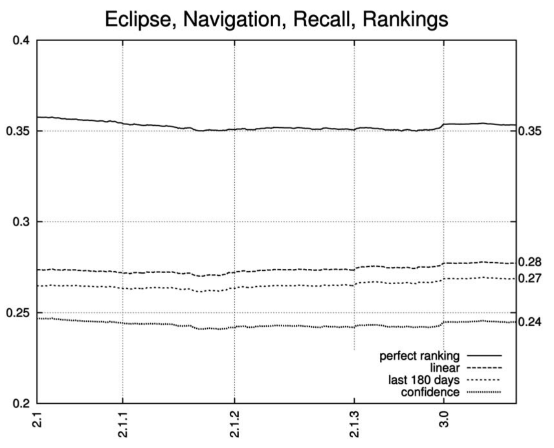


Fig. 9. Different ranking techniques for ECLIPSE (2003-03-28 to 2004-09-14).

7.11 Results: Recent Changes

So far, all changes to an item contribute equally to the support count and confidence, independently of when they occurred. Intuitively, recent changes are more likely to be relevant than older changes as they reflect the current state of the system or the current strategies in the project. As an example, consider *renaming*: If a method is renamed from m to m', the older changes to m are no longer relevant for new and future changes to m'.

Thus, we investigated whether we would improve recall and precision of the top 10 suggestions by ranking methods that take the *age of changes* into account. We explored two methods:

- **Last 180 days**. We computed the support count of an entity based on the last 180 days only and took this as ranking criterion. Thus, the position of an item in the list depends on how often it has been changed in the context of the situation within the last half year.
- **Linear weighting**. As an alternative, we used a linear increasing weighting function that assigns 0 to the oldest and 1 to the most recent transaction for a situation. As ranking criterion we took the sum of the weights for all transactions that support a suggestion.

As the ECLIPSE project is known to undergo frequent restructurings and renamings, we have focused on this particular project. Fig. 9 shows the evaluation results for ECLIPSE over time. We considered only queries for which ROSE returned more than 10 items because for the other queries, changing the ranking has no impact on recall and precision. The initial ranking criterion (confidence) yielded a recall of about 0.24. The "last 180 days" method improves the recall to 0.27; linear weighting enlarges the recall further to 0.28. Both ranking methods also increased the precision in similar fashion.

Weighting changes by age does not always improve results, though. We repeated the experiment with GCC and found precision and recall almost unchanged. This again can be attributed to the maturity and stability of GCC: If new changes are very similar to old changes, then assigning a higher weight to new changes will not make much difference.

To get a feeling for how much potential is covered at all by improving ranking, we computed the recall we would reach with *perfect ranking*—that is, the recall we can reach within the top 10 suggestions if all correct items are ranked to the top of ROSE's recommendations. In Fig. 9, we see that, for ECLIPSE, the difference between the recall reached with ranking by confidence and the recall for the perfect ranking is about 0.11; for GCC, we observed a similar room for improvement. This highlights the remaining potential, but also the limits of recommending related entities.

Summary. *For projects that are frequently restructured, assigning a higher weight to recent changes can increase precision and recall.*

7.12 Threats to Validity

We have studied more than 100,000 transactions of eight large 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, enforcing a strict change management would result in changes that are more logically separated; in ROSE, this yields a higher precision and higher recall—and, hence, a better predictability.

Transactions do not record the *order* of the individual changes involved because they have been committed simultaneously to the version archive. 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 item, 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.

8 Related Work

Independently from us, Ying et al. developed an approach that also uses association rule mining on CVS version archives [33]. They 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.

Xing and Stroulia [32] used association rule mining on versions of UML diagrams to detect class coevolution. Their approach yielded promising initial results, but has not yet been evaluated on a large scale.

Hassan and Holt investigated several heuristics to predict change propagation for fine-granular entities [16]. The heuristics have been based on historical cochange and static dependencies. Hassan and Holt did not consider association rules.

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 [22]. Various researchers computed metrics on the module or file level [3], [11], [14], [15] or orthogonal to these per feature [23] 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 [10]. The CVS history allows to detect more fine-grained logical coupling between classes [12], files, and functions [35]. None of these works on logical coupling did address its predictive power. Sayyad-Shirabad et al. used inductive learning to learn different concepts of relevance between logically coupled files [26], [27], [28]. 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. Sayyad-Shirabad et al. thoroughly evaluated the predictive power of the concepts found.

Michail used data mining on the source code of programming libraries to detect reuse patterns in form of association [20] or generalized association rules [21]. 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 [7] or program code [2]. HIPIKAT [8] improves on this by taking also other sources like mail archives and online documentation into account. All these tools focus on high recall (in contrast to ROSE, which focuses on high precision) and leverage relationships between files or classes (rather than between fine-grained entities).

9 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. In 63 percent of all transactions, ROSE makes a recommendation. These contain 45 percent of the related items, with a precision of more than 30 percent. In 90 percent of all recommendations, the three topmost suggestions contain a correct entity.
- For rapidly evolving systems like KOFFICE, 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.
- The predictive power of ROSE increases quickly at the start of a project; it is best during maintenance phases.
- In about 2-7 percent of all erroneous transactions, ROSE correctly detects the missing change. If such a warning occurs, it should be taken seriously, as only 2 percent 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" versus "New feature"). Furthermore, one can link changes to bug databases to identify *fix-inducing changes* [29]. Since such changes are later undone in a fix, ROSE should not use them for learning.
- **Refactorings**. Right now, ROSE does not recognize renamings of functions or files. We plan to integrate a detection of such refactorings [13] into ROSE that will result in additional rules for entities that have been renamed.
- **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 unlikely. Furthermore, coupling that can be found via program analysis [33] 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 [35], we used visual mining to detect regularities and irregularities of logically coupled items. Such visualizations could further explain the recommendations to programmers and managers.

ROSE is publicly available as a plug-in for ECLIPSE. For detailed 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. The development of ROSE as an ECLIPSE plug-in was made possible by an IBM ECLIPSE innovation grant. Holger Cleve, Carsten Görg, Christian Lindig, Stephan Neuhaus, Stefan Siersdorfer, and the anonymous reviewers gave helpful comments on earlier revisions of this paper.

References

[1] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proc. 20th Very Large Data Bases Conf. (VLDB)*, pp. 487-499, 1994.

[2] D.L. Atkins, "Version Sensitive Editing: Change History as a Programming Tool," *Proc. Conf. System Configuration Management (SCM '98)*, 1998.

[3] T. Ball, J.-M. Kim, A.A. Porter, and H.P. Siy, "If Your Version Control System Could Talk," *Proc. ICSE Workshop Process Modelling and Empirical Studies of Software Eng.*, 1997.

[4] B. Berliner, "CVS II: Parallelizing Software Development," *Proc. Winter 1990 USENIX Conf.*, pp. 341-352, Jan. 1990.

[5] J.M. Bieman, A.A. Andrews, and H.J. Yang, "Understanding Change-Proneness In OO Software through Visualization," *Proc. 11th Int'l Workshop Program Comprehension*, pp. 44-53, May 2003.

[6] M. Burch, S. Diehl, and P. Weißgerber, "Visual Data Mining in Software Archives," *Proc. ACM Symp. Software Visualization (SOFTVIS)*, 2005.

[7] A. Chen, E. Chou, J. Wong, A.Y. Yao, Q. Zhang, S. Zhang, and A. Michail, "CVSSearch: Searching through Source Code Using CVS Comments," *Proc. Int'l Conf. Software Methods*, pp. 364-374, 2001.

[8] D. Čubranić and G.C. Murphy, "Hipikat: Recommending Pertinent Software Development Artifacts," *Proc. Int'l Conf. Software Eng.*, pp. 408-418, 2003.

[9] K. Fogel and M. O'Neill, "cvs2cl.pl: CVS-Log-Message-to-ChangeLog Conversion Script," http://www.red-bean.com/cvs2cl/, Sept. 2002.

[10] H. Gall, K. Hajek, and M. Jazayeri, "Detection of Logical Coupling Based on Product Release History," *Proc. Int'l Conf. Software Maintenance (ICSM '98)*, pp. 190-198, Nov. 1998.

[11] H. Gall, M. Jazayeri, R. Klösch, and G. Trausmuth, "Software Evolution Observations Based on Product Release History," *Proc. Int'l Conf. Software Maintenance (ICSM '97)*, pp. 160-196, 1997.

[12] H. Gall, M. Jazayeri, and J. Krajewski, "CVS Release History Data for Detecting Logical Couplings," *Proc. Int'l Workshop Principles of Software Evolution*, pp. 13-23, 2003.

[13] C. Görg and P. Weißgerber, "Detecting and Visualizing Refactorings from Software Archives," *Proc. 13th Int'l Workshop Program Comprehension (IWPC)*, 2005.

[14] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," *IEEE Trans. Software Eng.*, vol. 26, no. 7, July 2000.

[15] A.E. Hassan and R.C. Holt, "The Chaos of Software Development," *Proc. Int'l Workshop Principles of Software Evolution*, 2003.

[16] A.E. Hassan and R.C. Holt, "Predicting Change Propagation in Software Systems," *Proc. Int'l Conf. Software Maintenance (ICSM 2004)*, Sept. 2004.

[17] *Proc. 25th Int'l Conf. Software Eng. (ICSE)*, May 2003.

[18] *Proc. Int'l Conf. Software Maintenance (ICSM 2001)*, Nov. 2001.

[19] *Proc. Int'l Workshop Principles of Software Evolution (IWPSE 2003)*, Sept. 2003.

[20] A. Michail, "Data Mining Library Reuse Patterns in User-Selected Applications," *Proc. 14th Int'l Conf. Automated Software Eng. (ASE '99)*, pp. 24-33, Oct. 1999.

[21] A. Michail, "Data Mining Library Reuse Patterns Using Generalized Association Rules," *Proc. Int'l Conf. Software Eng.*, pp. 167-176, 2000.

[22] A. Mockus and L.G. Votta, "Identifying Reasons for Software Changes Using Historic Databases," *Proc. Int'l Conf. Software Maintenance (ICSM 2000)*, pp. 120-130, Oct. 2000.

[23] A. Mockus, D.M. Weiss, and P. Zhang, "Understanding and Predicting Effort in Software Projects," *Proc. Int'l Conf. Software Eng.*, pp. 274-284, 2003.

[24] *Proc. Int'l Workshop Mining Software Repositories (MSR 2004)*, May 2004.

[25] C.J.V. Rijsbergen, *Information Retrieval*, second ed. London: Butterworths, 1979.

[26] J. Sayyad-Shirabad, T.C. Lethbridge, and S. Matwin, "Supporting Maintainance of Legacy Software with Data Mining Techniques," *Proc. Int'l Conf. Software Methods*, pp. 22-31, 2001.

[27] J. Sayyad-Shirabad, T.C. Lethbridge, and S. Matwin, "Mining the Maintenance History of a Legacy Software System," *Proc. Int'l Conf. Software Maintenance (ICSM 2003)*, Sept. 2003.

[28] J. Sayyad-Shirabad, T.C. Lethbridge, and S. Matwin, "Mining the Software Change Repository of a Legacy Telephony System," *Proc. Int'l Workshop Mining Software Repositories (MSR 2004)*, pp. 53-57, 2004.

[29] J. Śliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes? On Fridays," *Proc. Int'l Workshop Mining Software Repositories (MSR)*, May 2005.

[30] R. Srikant and R. Agrawal, "Mining Generalized Association Rules," *Proc. 21st Very Large Data Bases Conf. (VLDB)*, pp. 407-419, 1995.

[31] R. Srikant, Q. Vu, and R. Agrawal, "Mining Association Rules with Item Constraints," *Proc. Third Int'l Conf. KDD and Data Mining (KDD '97)*, Aug. 1997.

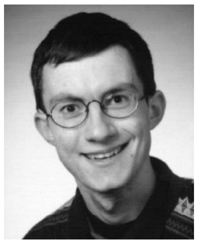
[32] Z. Xing and E. Stroulia, "Data-Mining in Support of Detecting Class Co-Evolution," *Proc. 16th Int'l Conf. Software Eng. and Knowledge Eng. (SEKE '04)*, June 2004.

[33] A.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll, "Predicting Source Code Changes by Mining Change History," *IEEE Trans. Software Eng.*, vol. 30, no. 9, pp. 574-586, Sept. 2004.

[34] T. Zimmermann, "Mining Version Archives to Guide Software Changes," Master's thesis, Universität Passau, Germany, June 2004.

[35] T. Zimmermann, S. Diehl, and A. Zeller, "How History Justifies System Architecture (or Not)," *Proc. Int'l Workshop Principles on Software Evolution*, pp. 73-83, 2003.

[36] T. Zimmermann and P. Weißgerber, "Preprocessing CVS Data For Fine-Grained Analysis," *Proc. Mining Software Repositories*, pp. 2-6, 2004.



Thomas Zimmermann received the diploma degree in computer science from the University of Passau in 2004. His diploma thesis involved the development of ROSE, which is presented in this paper. He is currently a PhD student at the Saarland University in Saarbrücken, supported by a scholarship from the DFG research training group on "Quality Guarantees for Computer Systems." His research interests are in software evolution, source code analysis, and data mining. He is student member of the IEEE and the IEEE Computer Society.

Peter Weißgerber received a diploma in computer science at Saarland University Saarbrücken in 2004. He is currently a PhD student and works as a research assistant in Prof. Stephan Diehl's reseach group at Catholic University Eichstätt. His main research interest is software evolution: How does software evolve over time and what can be learned from the past for the future? He is also interested in software visualization, especially in making the results of his research available to programmers.

Stephan Diehl received the PhD degree from Saarland University as a scholar of the German Research Foundation (DFG) working in the group of Prof. Reinhard Wilhelm. He is a professor of computer science at Catholic University Eichstätt. He studied computer science and computational linguistics at Saarland University and as a Fulbright scholar at Worcester Polytechnic Institute, Massachusetts. His research interests include programming languages and compiler design, Web technologies, and visualization, in particular, software visualization.

Andreas Zeller received the PhD degree at TU Braunschweig in 1997 and has been a tenured professor since 2001. He is a software engineering professor at Saarland University, Germany. He works on the analysis of large software systems and their history, especially the analysis of why these systems fail to work as they should. He is a member of the IEEE Computer Society.