UNIVERSITÄT PASSAU

Diploma Thesis

Mining Version Histories to Guide Software Changes

Thomas Zimmermann
tz@acm.org

Advisor:	Prof. Dr. Andreas Zeller
Referees:	Prof. Dr. Gregor Snelting
	Prof. Dr. Burkhard Freitag

Lehrstuhl für Softwaresysteme
Fakultät für Mathematik und Informatik
Universität Passau

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 coupling that is undetectable by program analysis, and

(c) prevent errors due to incomplete changes.

Our approach consists of two phases:

- *Preprocessing* mirrors a complete version history in a database, and searches for fine-grained changes—that are changes on functions rather than on complete files.
- *Mining* creates the rules that are used for recommendations. We developed our own mining technique that mines only for matching rules on the fly. Thus we can make up-to-date recommendations very fast.

Our *evaluation* involving eight large open-source projects shows that 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%.

Acknowledgments

A thousand thanks to *Prof. Andreas Zeller* for his supervision and continuous confidence in my work. He also introduced me to the software research community and funded my conference visits. Without the support of *Prof. Gregor Snelting* this thesis would never have been possible. He also agreed to be its first examiner. Very special thanks to him. Also, special thanks to *Prof. Burkhard Freitag* for being the second examiner. I am very happy to have these two examiners because both inspired me for software engineering and databases, respectively.

Very special thanks to *Prof. Stephan Diehl* and *Peter Weißgerber* for the great collaboration and the fruitful discussions. Thanks to the software evolution team at Saarland University, especially *Carsten Görg* and *Michael Burch*.

I owe to *Philipp Bouillon*, *Holger Cleve*, *Dr. Christian Lindig*, and *Stephan Neuhaus* their support, discussions, and their valuable computing power. They had to suffer a lot during the evaluation. Thanks to *Christian Langner* and *Christian Weber* for their 24/7 support in keeping my databases up. They are wonderful system administrators. Thanks to *Philipp Bouillon* and *Martin Burger* for sharing their exhaustive ECLIPSE knowledge.

Special thanks to *Annie Ying*, *Jennifer Bevan*, and *Jelber Sayyad-Shirabad* for the exchange of thoughts. Also special thank to *Ahmed Hassan*, *Richard Holt* and *Audris Mockus* for organizing the MSR 2004 workshop.

Special thanks to *Philipp Bouillon*, *Kathrin Feichtinger*, *Daniel Gmach*, *Paul Holleis*, *Richard Kuntschke*, *Stefanie Scherzinger*, and especially *Silvia Breu* for their valuable comments on this thesis and extensive proofreading. *Richard Kuntschke* provided the wonderful LaTeX template. Also, thanks to all proofreaders of former papers: *Holger Cleve*, *Stephan Diehl*, *Carsten Görg*, *Christian Lindig*, *Stephan Neuhaus*, *Stefan Siersdorfer*, *Peter Weißgerber*, *Andreas Zeller*, and the anonymous IWPSE, ICSE, and MSR reviewers. However, all mistakes are mine and hereby public domain.

Very special thanks to *Philipp Bouillon*, *Holger Cleve*, *Kerstin Reese*, and *Andreas Zeller* for providing me accommodation during some of my short visits in Saarbrücken. *Holger Cleve* helped me to find my own flat, and *Kerstin Reese* provided dishes and lots of other useful stuff for my household.

Miscellaneous thanks for entertainment reasons to: Lovely *Amanda Bynes*, *Greg Iles*, *Avril Lavigne*, *Michael Moore*, *Kathy Reichs*, and *Arnold Schwarzenegger*. Thanks to Sneakclub.com, especially *Stefan Bohlander*, *Michael Grundmann*, and *Marc Wilhelm* and to *Antalia Kebap*.

Finally, and most deeply, I thank my parents, *Veronika Zimmermann* and *Prof. Walter Zimmermann*, and my sister, *Andrea Winter*, for their loving support throughout my studies.

Contents

List of Figures

List of Tables

Table of Algorithms

Chapter 1

Introduction

Knowledge is a tool. Wisdom directs it.
– UniversalQuest.com

Shopping for the latest John Grisham novel "The Last Juror" at Amazon.com you may have come across a section called "Customers who bought this book also bought...". In this section, Amazon.com lists other books that are related by purchase with the current book. The aim is to point customers to other interesting books, and thereby increase the sales of Amazon.com. For our John Grisham book, the page recommends "Bleachers", "The Big Bad Wolf", "Split Second", "The Zero Game", and "3rd Degree" (see Figure 1.1 on the following page). Such information is gathered by *data mining*—the automated extraction of hidden predictive information from large data sets, e.g., the purchase history of Amazon.com customers.

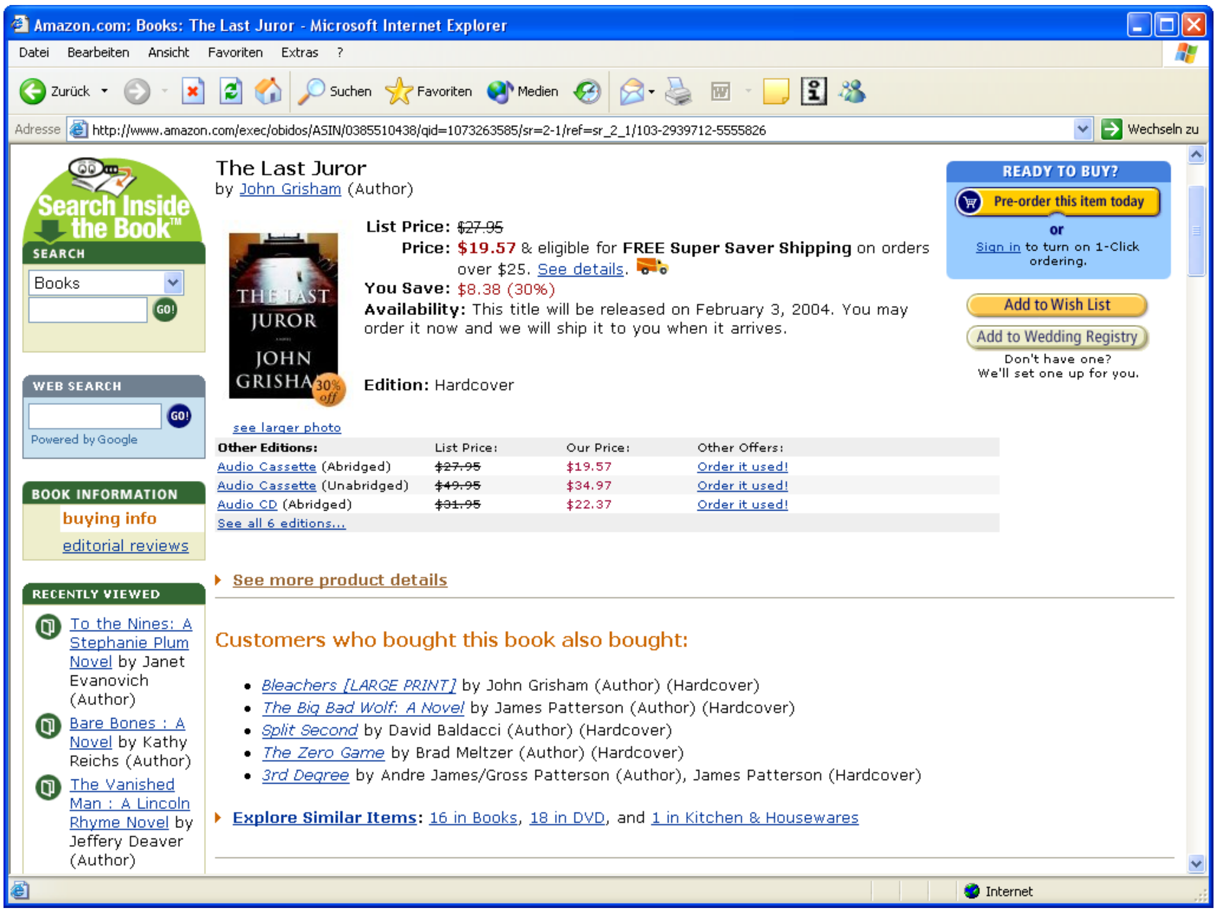
This feature is not restricted to books; it can be applied to any kind of product or data. Amazon.com provides such information for CDs, DVDs, electronics, toys, and games. About two years ago, Amazon.com even recommended shoppers of "Essential .NET, Volume 1" to wear "Clean Underwear" which turned out to be a random recommendation; Amazon.com just wanted to promote its new apparel shop [All02, Wag02].

Alexa.com Internet behavior of users of its toolbar. This allows Alexa.com to make statements like "People who visit CNN.com Interactive also visit, among other pages, USA Today or The Weather Channel". It even connects Internet sites with books, e.g., it recommends visitors of the Scholastic Publishing Corporation to buy Harry Potter books.[1]

The examples above show that data mining techniques have become a day-to-day part of e-commerce and are now essential for increasing the performance of a business. However, data mining is not restricted to e-commerce. It can be used everywhere—for example in *program analysis*.

Generally, two forms of program analysis exist: *static* and *dynamic* analysis. Static analysis does not execute any programs and mainly uses deduction as a reasoning technique. In contrast, dynamic analysis relies on observation, induction, and experimentation, and therefore executes programs to gather dynamic data [Zel03].

[1]Actually Scholastic Publishing Corporation is the publisher of Harry Potter books.


Amazon.com: Books: The Last Juror - Microsoft Internet Explorer
Datei Bearbeiten Ansicht Favoriten Extras ?
Zurück Suchen Favoriten Medien
Adresse http://www.amazon.com/exec/obidos/ASIN/0385510438/qid=1073263585/sr=2-1/ref=sr_2_1/103-2939712-5555826 Wechseln zu
Search Inside the Book™
SEARCH
Books
GO!
WEB SEARCH
GO!
Powered by Google
BOOK INFORMATION
buying info
editorial reviews
RECENTLY VIEWED
READY TO BUY?
Pre-order this item today
or
Sign in to turn on 1-Click ordering.
Add to Wish List
Add to Wedding Registry
Don't have one? We'll set one up for you.
Internet

The Last Juror
by John Grisham (Author)

List Price: ~~$27.95~~
Price: $19.57 & eligible for **FREE Super Saver Shipping** on orders over $25. See details.
You Save: $8.38 (30%)
Availability: This title will be released on February 3, 2004. You may order it now and we will ship it to you when it arrives.

Edition: Hardcover

see larger photo

Other Editions:	List Price:	Our Price:	Other Offers:
Audio Cassette (Abridged)	~~$27.95~~	$19.57	Order it used!
Audio Cassette (Unabridged)	~~$49.95~~	$34.97	Order it used!
Audio CD (Abridged)	~~$31.95~~	$22.37	Order it used!
See all 6 editions...			

See more product details

Customers who bought this book also bought:

- *Bleachers [LARGE PRINT]* by John Grisham (Author) (Hardcover)
- *The Big Bad Wolf: A Novel* by James Patterson (Author) (Hardcover)
- *Split Second* by David Baldacci (Author) (Hardcover)
- *The Zero Game* by Brad Meltzer (Author) (Hardcover)
- *3rd Degree* by Andre James/Gross Patterson (Author), James Patterson (Hardcover)

Explore Similar Items: 16 in Books, 18 in DVD, and 1 in Kitchen & Housewares

To the Nines: A Stephanie Plum Novel by Janet Evanovich (Author)
Bare Bones : A Novel by Kathy Reichs (Author)
The Vanished Man : A Lincoln Rhyme Novel by Jeffery Deaver (Author)

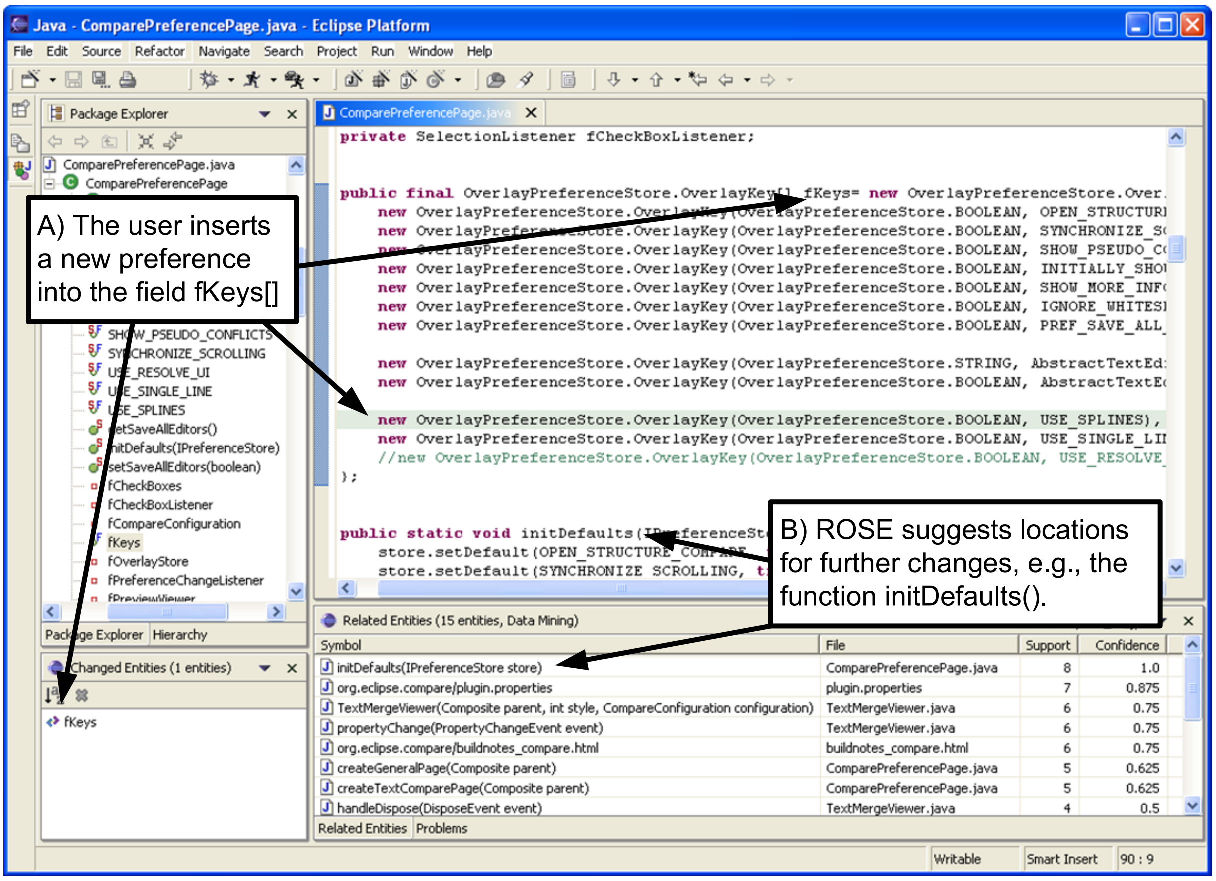
Figure 1.1: Customers who Bought this Book also Bought…

In practice, most program analysis techniques only consider a single version of the program, thus neglecting one very large source of information: the *version archive*. It contains a vast amount of information: Who changed what, why, and when? Usually, *software evolution* analyzes such data, but it is also valuable input for program analysis, both static and dynamic.

To some extent a version archive is similar to Amazon.com: As customers buy books, programmers make changes, and both do it in transactions. So it is only a matter of time until users will request a "Developers who changed this also changed…"-feature in their favorite IDE. The realization of such a feature is the topic of this diploma thesis.

The main part of the thesis is about the ROSE tool. ROSE is an acronym for *Reengineering Of Software Evolution* and is *not* related to Rational Rose. It analyzes version histories and guides programmers along related changes. To accomplish this, it uses two different types of recommendations, both similar to those of Amazon.com:

- *"Programmers who changed the selected function, also changed…"*
  In order to provide this information, ROSE performs mining only with respect to the currently selected or edited function. This approach corresponds directly to the "Customers who bought this book also bought…"-list of Amazon.com that is displayed on book pages, like the one for "The Last Juror" in Figure 1.1.


Java - ComparePreferencePage.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
ComparePreferencePage.java
ComparePreferencePage
A) The user inserts a new preference into the field fKeys[]
SHOW_PSEUDO_CONFLICTS
SYNCHRONIZE_SCROLLING
USE_RESOLVE_UI
USE_SINGLE_LINE
USE_SPLINES
getSaveAllEditors()
initDefaults(IPreferenceStore)
setSaveAllEditors(boolean)
fCheckBoxes
fCheckBoxListener
fCompareConfiguration
fKeys
fOverlayStore
fPreferenceChangeListener
fPreviewViewer
Package Explorer
Hierarchy
Changed Entities (1 entities)
fKeys
ComparePreferencePage.java
B) ROSE suggests locations for further changes, e.g., the function initDefaults().
Related Entities (15 entities, Data Mining)
Related Entities
Problems
Writable
Smart Insert
90 : 9

```
private SelectionListener fCheckBoxListener;


public final OverlayPreferenceStore.OverlayKey[] fKeys= new OverlayPreferenceStore.Over
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, OPEN_STRUCTURI
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, SYNCHRONIZE_S(
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, SHOW_PSEUDO_C(
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, INITIALLY_SHO
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, SHOW_MORE_INF(
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, IGNORE_WHITES
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, PREF_SAVE_ALL_

    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.STRING, AbstractTextEd:
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, AbstractTextE(

    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, USE_SPLINES),
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, USE_SINGLE_LI
    //new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, USE_RESOLVE_
};


public static void initDefaults(IPreferenceSt
    store.setDefault(OPEN_STRUCTURE_COMPARE,
    store.setDefault(SYNCHRONIZE_SCROLLING, t
```

Symbol	File	Support	Confidence
initDefaults(IPreferenceStore store)	ComparePreferencePage.java	8	1.0
org.eclipse.compare/plugin.properties	plugin.properties	7	0.875
TextMergeViewer(Composite parent, int style, CompareConfiguration configuration)	TextMergeViewer.java	6	0.75
propertyChange(PropertyChangeEvent event)	TextMergeViewer.java	6	0.75
org.eclipse.compare/buildnotes_compare.html	buildnotes_compare.html	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 1.2: Programmers who Changed this Function also Changed. . .

- *“Programmers who changed functions you have changed, also changed . . . ”*
  In this case, ROSE records all changes and performs data mining with respect to these changes. This feature corresponds to the Amazon.com “Customers who bought items in your Shopping Cart also bought. . . ”-list that is displayed before you place your order. Obviously, the aim of such a list is to avoid that customers miss a relevant item; or, specific to ROSE, to avoid that programmers forget to change a relevant function.

At a first glance, ROSE seems to be nothing more than a nice-to-have feature that suggests likely changes. However ROSE is much more: Besides its navigation abilities, it can prevent errors due to incomplete changes and even reveals dependencies undetectable by program analysis. These features are illustrated below.

Suggest and predict likely changes. Suppose you are a programmer and just made a change. What else do you have to change? Figure 1.2 shows the ROSE tool as a plug-in for the ECLIPSE programming environment. The programmer is extending ECLIPSE with a new preference, and has added an element to the fKeys[] array. In the *What*s Related* view, ROSE now suggests to consider further changes, as inferred from the ECLIPSE version history. On top of the list are locations with highest *confidence*—that is, the likelihood that further changes should be applied to the presented location.

Prevent errors due to incomplete changes. In Figure 1.2, the top location, initDefaults(), 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 a programmer now wanted to commit changes *without* altering the suggested location, ROSE would issue a warning. (This warning is shown in Figure 6.3 on page 65.)

Reveal coupling undetectable by program analysis. As ROSE operates uniquely on the version history, it is able to find coupling between items that cannot be detected by program analysis—including coupling between items that are not even programs. In Figure 1.2, position 3 on the list is an ECLIPSE HTML documentation file with a confidence of 0.75, and suggests that after adding the new preference, the documentation should be updated, too.

Contributions

ROSE is not the first tool to leverage version histories. In earlier work researchers have used history data to understand programs and their evolution [BKPS97], to detect evolutionary coupling between files [GHJ98] or classes [BAY03], or to support navigation in the source code [ČM03]. In contrast to this state of the art, the presented work

- gives a detailed overview about *preprocessing* techniques,
- uses fully-fledged *data mining* to obtain association rules from version histories,
- detects coupling between fine-grained *program entities* such as functions or variables (rather than, say, classes), thus increases precision and integrates with program analysis,
- thoroughly evaluates the *ability to predict future or missing changes,* thus evaluating the actual usefulness of our techniques, and
- provides a prototype implementation of the presented techniques.

For a full discussion of related work see Chapter 7. Parts of this thesis have been published and presented at

- the *International Conference on Software Engineering* (ICSE 2004) [ZWDZ04],
- the *International Workshop on Mining Software Repositories* (MSR 2004) [ZW04], and
- the *International Workshop on Principles of Software Evolution* (IWPSE 2003) [ZDZ03].

ROSE has been awarded with an *IBM Eclipse Innovation Grant* [IBM04]. It will be available for download in Fall 2004.

Structure

The remainder of this thesis is organized as follows. Chapter 2 gives a brief *overview* on ROSE and the applied techniques. Chapter 3 discusses the two preprocessing steps of ROSE: *data collection*, which is the transformation of a version archive into a database, and *data cleaning*, which is the identification of outliers. Chapter 4 presents details about the mining process. It introduces rules, simple measures for rule assessment, and *data mining* algorithms. Chapter 5 gives more examples for rules. Chapter 6 covers the *evaluation* of ROSE. It describes the evaluation setup and measures, and discusses the results for the navigation and error prevention abilities of ROSE. Chapter 7 gives an overview of *related work* and Chapter 8 concludes with the presentation of *future work*.

Chapter 2

Overview

2.1 CVS in a Nutshell

Large software projects are constantly evolving under the influence of many programmers. Therefore, it is very important to track changes and to coordinate developers. In practice, *version control systems* facilitate these tasks. An example for such a tool is the *Concurrent Versions System (CVS)*, used by most open source projects [CVS04].

CVS uses two concepts called the *repository* and the *working directory*. The repository (or *version archive*) contains all information required to restore historical versions of any file in the project. A developer fetches the latest code from the repository to his working directory, makes his changes, and then commits those changes back to the repository.

Each changed file is stored as a version in the repository. CVS uses revision numbers (e.g., 1.42) to distinguish between different versions of a file. The most recent revision is called the *head* revision. It is also possible to create symbolic names for revisions. These *tags* are often used to mark releases or important milestones.

We refer to a commited file as a *checkin*, and to a whole commit operation as a *commit*. Therefore, a commit is simply composed of at least one checkin. CVS commits all files individually which means, that it does not track commits and we have to recover them for our analysis. For that purpose, we can use additional information that CVS stores for a checkin:

- The *author*, i.e., the user name of the programmer who committed the change;
- The *extent*, i.e., the file and location affected by the change;
- The *content*, i.e., the actual text or data inserted, deleted, or modified;
- The *rationale*, i.e., the reason why the change was made;
- The *date* of the checkin.

We refer to a recovered commit as a *transaction*. Do not expect CVS commits or transactions to fulfill the ACID paradigm [HR83]: atomicity, consistency, and isolation are not guaranteed be-

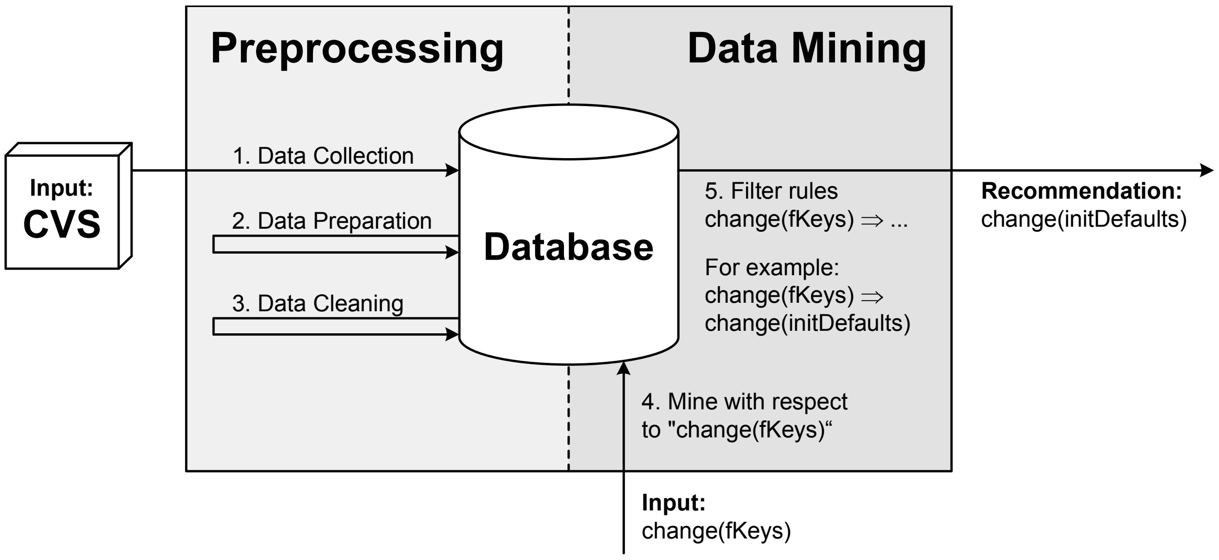


Figure 2.1: ROSE Overview

cause commits are carried out consist of single checkins, and only single directories are locked. Durability holds if all checkins are written directly to disk.

Often a linear development process is not sufficient for larger software projects. Therefore, CVS allows to create separate development lines called *branches*. A branch originates from another branch or the main development line. Branches are frequently used for experimental implementations or for bug-fixes. Most branches integrate their changes back to the original development line at some future point. This event is called *merging* and carried out as one large transaction. This transaction simply reproduces changes made on the branch. Unfortunately, they are not marked in the CVS archives.

2.2 A Guided Tour of ROSE

Figure 2.1 illustrates the workflow of ROSE. It splits into 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 allows a fast access to all necessary information.

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

In order to become familiar with ROSE, we present the individual steps using a small example. More details about preprocessing and mining are described in Chapters 3 and 4.

2.2.1 Data Collection

Suppose, there are three developers (Harry, Hermione, and Ron), who work on a small project[1] consisting of three files (AB.java, C.java, and D.java). Harry, Hermione, and Ron use CVS to archive their versions. CVS stores the necessary data in a very cryptic way, and access to the data is rather slow and complicated. So, the obvious solution is to collect all information in a database. Figure 2.2(a) on the following page shows the result of this step.

2.2.2 Data Preparation

The collected data is not suitable for data mining, yet. First, CVS has no transaction concept, and second, CVS provides changes on file level only.

Step 1: Group Checkins to Transactions

Thus, the first task is to group checkins to transactions. A transaction consists of several checkins that all have the same author, log message, and timestamp. In Figure 2.2(b), the checkins AB.java, revision 1.47, and D.java, revision 1.42, have both been made by Ron on 2004-01-05 with the log message "Changed this and that". Thus, they belong to the same transaction #1.

Step 2: Map Changes to Entities

Basically, the data is now ready for data mining. However, since we have only changes and transactions on file level, we only can make statements about files. Thus, the next step is to increase the granularity and to find the changes inside a file. We introduce the concept of *entities* to formalize that. An entity is a syntactic component of a file. Possible entities for source code are for example classes, functions, and declarations. A checkin can affect multiple entities: For instance, in Figure 2.2(c) the checkin of AB.java, revision 1.48, changed two entities: accio() and banish().

2.2.3 Data Cleaning

Usually, all checkins of a transaction are related to each other. In some cases there are exceptions. For instance, in CVS, a *merge* of two branches simply applies changes made on one branch to the other. This is noise for two reasons: Changes (and relations) on branches are overrated, and additional (wrong) relations are introduced because all transactions made on the branch are summarized into *one* merge transaction. Data cleaning identifies such transactions and removes them. In Figure 2.2(c), transaction #3 has been identified as noise and will not be considered for data mining.

[1]Actually, the project is for their Muggle Studies class and covers ancient version control systems.

Changed File	Rev.	Date	Autor	Log Message
AB.java	1.47	2004-01-05	Ron	Changed this and that
AB.java	1.48	2004-01-06	Hermione	More changes
AB.java	1.49	2004-01-06	Harry	And a merge
AB.java	1.50	2004-01-08	Ron	Some changes
AB.java	1.51	2004-01-10	Hermione	Fixed a bug
C.java	1.23	2004-01-06	Hermione	More changes
C.java	1.24	2004-01-06	Harry	And a merge
C.java	1.25	2004-01-10	Hermione	Fixed a bug
D.java	1.42	2004-01-05	Ron	Changed this and that
D.java	1.43	2004-01-06	Harry	And a merge

(a) Data Collection: Extract CVS Information

Transaction	Changed File	Rev.	Date	Autor	Log Message
#1	AB.java	1.47	2004-01-05	Ron	Changed this and that
	D.java	1.42	2004-01-05	Ron	Changed this and that
#2	AB.java	1.48	2004-01-06	Hermione	More changes
	C.java	1.23	2004-01-06	Hermione	More changes
#3	AB.java	1.49	2004-01-06	Harry	And a merge
	C.java	1.24	2004-01-06	Harry	And a merge
	D.java	1.43	2004-01-06	Harry	And a merge
#4	AB.java	1.50	2004-01-08	Ron	Some changes
#5	AB.java	1.51	2004-01-10	Hermione	Fixed a bug
	C.java	1.25	2004-01-10	Hermione	Fixed a bug

(b) Data Preparation: Group Changes to Transactions

Transaction	Changed File	Rev.	Changed Symbol	Log Message
#1	AB.java	1.47	banish()	Changed this and that
	D.java	1.42	deletrius()	Changed this and that
#2	AB.java	1.48	accio()	More changes
	AB.java	1.48	banish()	More changes
	C.java	1.23	**confundus()**	More changes
#3	AB.java	1.49	accio()	And a merge
	AB.java	1.49	banish()	And a merge
	C.java	1.24	confundus()	And a merge
	D.java	1.43	deletrius()	And a merge
#4	AB.java	1.50	accio()	Some changes
	AB.java	1.50	banish()	Some changes
#5	AB.java	1.51	accio()	Fixed a bug
	AB.java	1.51	banish()	Fixed a bug
	C.java	1.25	**confundus()**	Fixed a bug

(c) Data Preparation: Map Changes to Entities

Figure 2.2: Data Preprocessing: Build the Database

2.2.4 Data Mining

Once preprocessing is completed, the data can be used for data mining, for example for *association rule* mining. An association rule simply expresses the relation between at least two entities, e.g., "If you have changed fKeys[], then change initDefaults(), too.", or more formally "changed(fKeys[]) $\Rightarrow$ change(initDefaults())".

Such rules are generated using the transactions of the preprocessed version data. In practice, a rule is not valid for all transactions. Three measures express the importance of a single rule:

- *Frequency* is the number of transactions for which the rule was valid.
- *Support* is the frequency related to the total number of transactions.
- *Confidence* is the likelihood that the rule holds if the left side is satisfied.

ROSE works on very fine-granular entities to detect such rules. This means that, if possible, it concentrates on functions and variables rather than on files or modules. This high granularity results in very precise rules, and thus in a high locality for the recommendations of ROSE. Often coarse-grained rules are misleading because they summarize many fine-grained rules in one coarse-grained rule.

Mine for all Rules

ROSE can search for all rules that have a minimum frequency (or support) and a minimum confidence. Such a search may reveal hidden dependencies and is of special interest to managers or project leaders. For our example, Figure 2.3 on the next page shows all rules with a minimum frequency of 2 and a minimum confidence of 50%.

Mining for all rules reveals general patterns and hot-spots, and has been the topic of another diploma thesis in this research project [Wei04].

Mine for Rules with Constraints

As Figure 2.3 indicates, there are many possible rules. In practice, very high frequency and confidence thresholds are required to get a manageable number of rules. However, if we increase the minimum frequency in our example from 2 to 3, we will get only the first two rules instead of all twelve rules of Figure 2.3. However, fewer rules also mean fewer *coverage* of entities. In our case, the first two rules only cover accio() and banish(), but not confundus(). In other words, ROSE cannot make any recommendations about confundus().

In order to get a high coverage, ROSE mines for association rules *on demand*. This has the advantage, that it can use information about user changes as an input for the mining algorithm—or, more precisely, as a constraint for the left-hand side of a rule.

Suppose, Hagrid is new to our project and changes confundus(). ROSE now only considers transactions that contain confundus()—in our case transactions #2 and #5 of Figure 2.2(c).

Rule	Confidence	Support
change(accio()) ⇒ change(banish())	1.00	3
change(banish()) ⇒ change(accio())	0.75	3
change(accio()) ⇒ change(confundus())	0.67	2
change(confundus()) ⇒ **change(accio())**	1.00	2
change(banish()) ⇒ change(confundus())	0.50	2
change(confundus()) ⇒ **change(banish())**	1.00	2
change(accio()) ⇒ change(banish()) ∧ change(confundus())	0.67	2
change(banish()) ⇒ change(accio()) ∧ change(confundus())	0.50	2
change(confundus()) ⇒ **change(accio())** ∧ **change(banish())**	1.00	2
change(accio()) ∧ change(banish()) ⇒ change(confundus())	0.67	2
change(accio()) ∧ change(confundus()) ⇒ change(banish())	1.00	2
change(banish()) ∧ change(confundus()) ⇒ change(accio())	1.00	2

Figure 2.3: Data Mining: Create Association Rules

Because the number of transactions and entities is reduced, creating association rules becomes a very cheap operation. The resulting rules are boldfaced in Figure 2.3.

This approach and the techniques to speed up the mining procedure are part of this thesis. They are described in detail in Chapter 4.

Chapter 3

Preprocessing

If your version control system could talk...
– Tom Ball et al. [BKPS97]

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

So, why is CVS so silent? In this chapter we address four limitations of CVS and present preprocessing techniques dealing with those issues:

1. *CVS has limited query functionality and is slow.*
   As mentioned above accessing information other than log information for single files is difficult in CVS. Furthermore, access is very slow because CVS uses the RCS file format. The obvious solution is to copy the whole CVS repository into a database. Thus, a multitude of queries are enabled and can be evaluated very fast.

2. *CVS splits up changes on multiple files into single checkins.*
   If a developer commits several files simultaneously, CVS checks them in individually discarding the relations between them. As ROSE relies on such relations, we have to infer transactions. Usually, a transaction corresponds to exactly one commit operation.

3. *CVS knows only files—but what about changes on functions?*
   The usefulness of ROSE depends on the granularity of its recommendations. This granularity is restricted by CVS to the file level. We need to analyze changes and detect the affected fine-grained entities in order to suggest functions or declarations.

4. *CVS contains unreliable data.*
   Some specialties of CVS call for data cleaning. For instance, merges or imports falsify the results of ROSE.

Many of these problems are specific to the analysis of CVS archives; more sophisticated version control systems, like SUBVERSION [Sub04], require less data preprocessing.

Note that all preprocessing steps can also be done *incrementally*—it is only necessary to preprocess data for new revisions instead of working on the whole repository again. To determine new revisions several approaches exist: Many open-source projects send an email to a mailing list for each commit. This approach is based on the *commitinfo* and *loginfo* files that can also be used to track commits on the server-side. A possibility to get recently changed files on the client-side is the CVS *rdiff* operation (with option *-s* for *summary*), or the CVS *status* operation.

3.1 Definitions

In this section we introduce formal definitions for changes, checkins[1], commits, transactions, and entities, generalizing the concepts found in existing version archives.

Adopting the notation from [ZH02], a *change* or *checkin* is a mapping $\delta : \mathcal{P} \to \mathcal{P}$, which, when applied, transforms a product $p \in \mathcal{P}$ into a *changed product* $p' = \delta(p) \in \mathcal{P}$. 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 *commits* or *transactions* that consist 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 $\Delta_{1,2}(p_1) = (\delta_1 \circ \delta_2 \circ \cdots \circ \delta_n)(p_1) = \delta_1\big(\delta_2\big(\cdots \delta_n(p_1)\big)\big) = p_2$.

The difference between *commits* and *transactions* is that a commit refers to the user activity (performing a CVS commit operation), while ROSE uses transactions to abstract from commits. Ideally, a transaction corresponds to exactly one commit, but we will see later that this is not always possible.

To express all syntactic components affected by a change, we define the concept of *entities*. An entity is a triple (c, i, p), where i is the identifier of the affected component, c is the syntactic category such as *method*, *class*, or *file*, and p is the parent component, or $\perp$ if there is none. A simple example for an entity is $(\textit{method}, \textit{f()}, (\textit{class}, \textit{Foo}, (\textit{file}, \textsf{Foo.java}, \perp)))$.

As a short notation for an entity we simply use i if the category c and the parent entity p are known in the context, or if i is unique on its own. This notation is frequently used for the parent entity, for instance $(\textit{method}, \textit{f()}, Foo)$.

The parent p is needed to distinguish between entities that have the same identifier i. For instance, we have two classes *Foo* and *Bar* that both contain a method called *f()*. Then we create two different entities: $(\textit{method}, \textit{f()}, \textit{Foo})$ and $(\textit{method}, \textit{f()}, \textit{Bar})$.

The mapping *entities* retrieves all entities affected by a change, checkin, or transaction. For instance, transaction #4 of our example from Figure 2.2(b) changes accio() and banish() in class

[1]In the style of CVS, we spell check-in as checkin.

AB of file AB.java. Thus, *entities* retrieves:

$$\mathit{entities}(\Delta) = \mathit{entities}(\delta_1) \cup \cdots \cup \mathit{entities}(\delta_n) = \left\{ \begin{array}{lll} (\mathit{method}, & \text{accio()}, & \text{AB}), \\ (\mathit{method}, & \text{banish[]}, & \text{AB}), \\ (\mathit{class}, & \text{AB}, & \text{AB.java}), \\ (\mathit{file}, & \text{AB.java}, & \perp) \end{array} \right\}$$

Entities are the base for mining: "I changed one entity; which other entities should I typically change?" We will not use all syntactic categories for mining. In the remainder we concentrate on the fine-granular entities, like methods or fields. Only if it is impossible to detect fine-grained changes, we will use coarse-grained entities, e.g., for plugin.properties.

For simplicity we introduce a second notation for transactions. A transaction T consists of several items. An *item* is an entity e combined with an *action*, e.g., *change* the entity e. As this notation will be used in our mining approach, we only include relevant entities, i.e., for no entity e its parent entity is included in T:

$$T = \{\text{change}(\text{accio()}), \text{change}(\text{banish[]})\}$$

The function *author* returns the programmer who committed the changes, *log_message* her rationale, and *time* the timestamp when she committed her changes. For transaction #4 from Figure 2.2(b) on page 10 the results are:

$$\begin{aligned} \mathit{author}(\Delta) &= \text{“Ron”} \\ \mathit{log\_message}(\Delta) &= \text{“Some changes”} \\ \mathit{time}(\Delta) &= \text{“2004-01-08 11:42:12 a.m.”} \end{aligned}$$

The function *time*(Δ) is ambiguous as transactions are not atomic in CVS. Thus, we define *begin_time* and *end_time* that give us the timestamps of the begin and end of a transaction, respectively.

3.2 Extract Information from CVS

First of all, we extract all information contained in a CVS archive into a database. This enables fast access to all data required in the following steps. Figure 3.1 on page 17 shows the database schema, which consists of several tables:

Directories are identified with *DirectoryName* and have an additional attribute *Depth*, that gives the level in the directory tree. For instance, a directory named foo/ has depth 1.

Files are identified with the attribute *FileID*. The table *Files* stores the fully qualified file name in *QualifiedFileName*. Additionally, this name is split into a *DirectoryName* and a *FileName*. This redundancy improves access to files because it is now possible to create indexes on parts of the fully qualified file name.

Furthermore, *DirectoryName* is a foreign key to table *Directories*. Using this relation, the depth of a file can be determined very fast by addition of one to the depth of the surrounding directory.

Both attributes *FileExtension* and *KeywordExpansion* are optional. *FileExtension* enables fast access to the file extension without parsing complete file names, and *KeywordExpansion* indicates whether a file is binary or not.

Checkins represent all revisions contained in a CVS archive. A checkin affects exactly one file expressed by *FileID*, which is a foreign key to table *Files*. The primary key of this table is *FileID* together with *RevisionID* because one file can only have one revision number once. The attribute *CheckinTime* contains the timestamp when the checkin was made.

As checkins are part of transactions, there is another foreign key called *TransactionID* to table *Transactions*. The description of a checkin is stored in table *Transactions* because it is shared with the other checkins of the transaction.

The optional attributes *Plus* and *Minus* give the number of modified lines; *State* marks whether the file was changed ("Exp") or removed ("dead"); and *BranchPrefix* contains a reference to the branch on which the checkin was made. *BranchPrefix* is empty if it is the main branch.

Transactions are identified by *TransactionID*. A transaction is committed by an *Author* who describes her rationale in a *Message*. Unfortunately, transactions are not atomic in CVS. Therefore, we have two timestamps: *BeginTime* and *EndTime*. The flag *IsNoise* indicates whether a transactions is relevant for mining or not.

One characteristic of table *Transactions* is the attribute *MessageMD5*. Many database systems provide only limited functionality for fields of unrestricted size like *Message*. For instance, Microsoft SQL Server can only group fields with a maximum size of 8,060 bytes [Mic04]. But the remaining analysis needs to group an unrestricted *Message* field. As a workaround, we created *MessageMD5* which contains the MD5 [Riv92] encrypted content of *Message*, and can be grouped by any database system.

Tags are used to mark particular revisions with symbolic names. The revision is identified by a foreign key: *FileID* and *RevisionID*. The symbolic name is saved in *TagName*. A revision can have more than one tag, but each tag can only be used once for a file. Thus, the primary key of *Tags* is the tuple (*FileID*, *TagName*).

Branches are created on file level in CVS. Each branch has a branch prefix that all revisions on the branch have in common. This *BranchPrefix* determines the revision where the branch originated. Therefore, *FileID* and *BranchPrefix* are the primary key. The revision of which the branch originated is implemented as a foreign key (*FileID*, *OriginRevision*) to table *Files*. In CVS, each branch gets an internal revision, *InternalRevision*, and a public symbolic name, *BranchName*.

The extraction process is straightforward (see Algorithm 3.1). Let F be the set of files that will be extracted. We insert each file f into table *Files*. Then we call the CVS *log* command and parse its output. Figure 3.2 on page 19 illustrates this parsing step. Next, we insert all

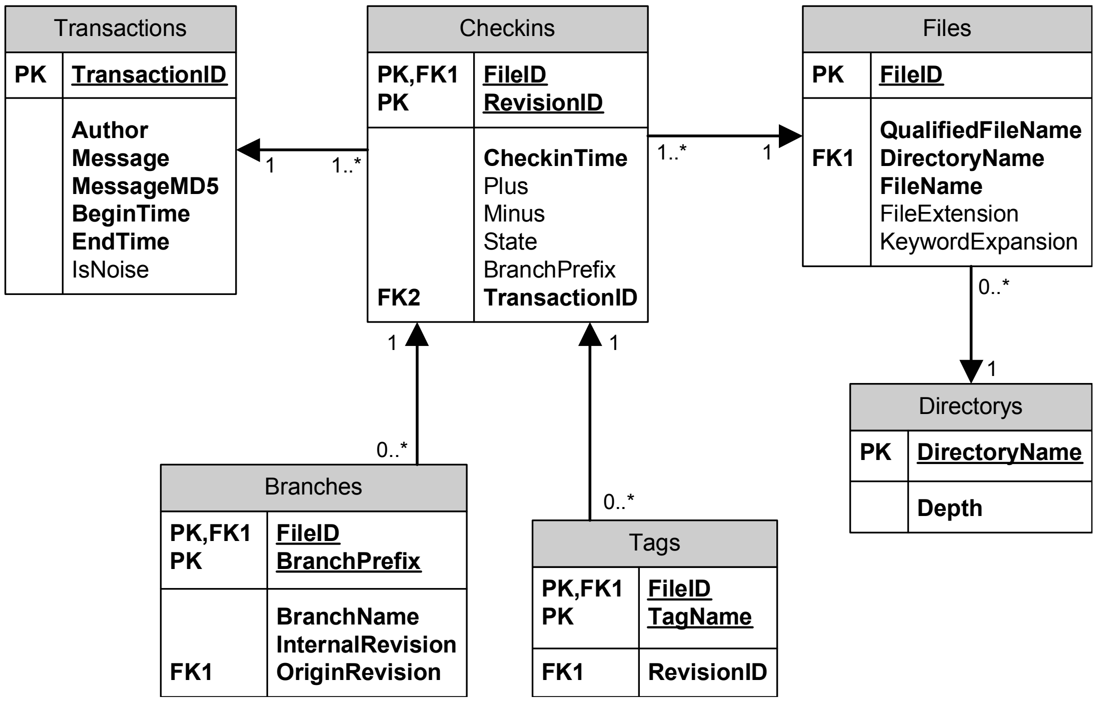


Figure 3.1: ROSE Database Schema: Tables for CVS

revisions into tables *Checkins* and *Transactions*, all symbolic names into table *Tags*, and all branch information into table *Branches*.

There exists one characteristic of the extraction: Each checkin is considered as a single transaction. After the extraction is completed, checkins will be grouped to larger transactions (see Section 3.3). Basically, it is possible to integrate grouping into extraction. But separation of both tasks is easier and faster because the number of total scans of *Checkins* and *Transactions* is reduced.

Branches require a special treatment during extraction because all necessary data is spread across several locations in the CVS *log* output. The "symbolic names"-part contains all *InternalRevision*s and *BranchName*s. An internal revision is identified with the included zero. For instance, 1.15.0.2 is internal and belongs to table *Branches*, 1.15 is a regular symbolic name and is stored in table *Tags*. The "revision"-blocks contain all remaining data for branches, like *BranchPrefix* and *OriginRevision*. The link between an *InternalRevision* and a *BranchPrefix* is established by removing "0." from the *InternalRevision*. For example, an internal revision 1.15.0.2 is connected to a branch prefix 1.15.2.

Another important aspect is the selection of files F—extract all files, or only files on the main development line? And what about deleted files? These decisions have a direct impact on the quality of the results. Thus, they depend heavily on the application:

Software evolution analysis. As the keyword "evolution" suggests, the *whole* history is of interest. This includes *all* files, no matter which development line. Even deleted files are important because they also represent evolution. The CVS operation *log* called with no parameter returns all files that ever existed.

Algorithm 3.1 CVS Extraction Algorithm

Input: Files F, Database $\mathcal{D}$

Output: Database $\mathcal{D}$

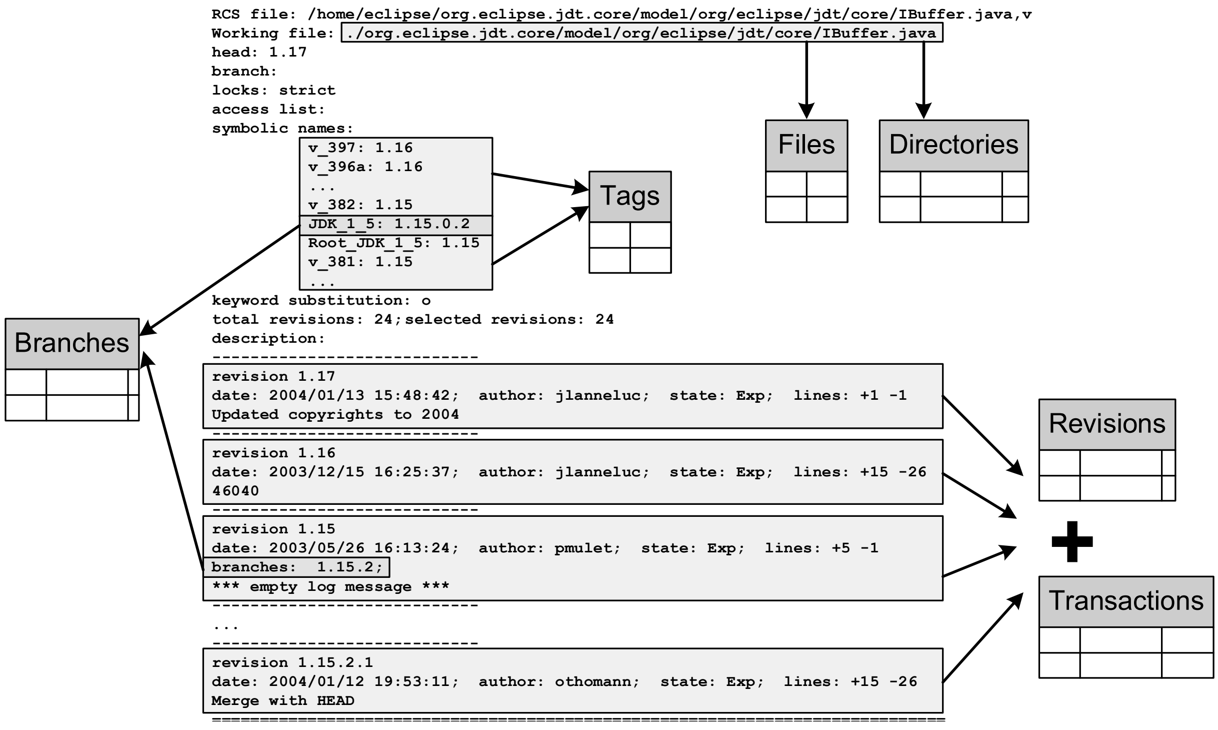
```
procedure EXTRACT($F$)
    $fileid = 0$
    $tid = 0$
    for all files $f$ in $F$ do
        if directory $d$ of $f$ is not in table Directories then
            insert $d$ into table Directories
        end if
        $fileid = fileid + 1$
        insert fileid, $f$ into table Files
        call CVS log for $f$
        parse output for revisions $R$, tags $T$, and branches $B$
        for all revisions $r$ in $R$ do
            $tid = tid + 1$
            insert $r$, fileid, timestamp and tid into table Checkins
            insert tid, author, and message into table Transactions
        end for
        for all tags $t$ in $T$ do
            insert data of t into table Tags
        end for
        for all branches $b$ in $B$ do
            insert data of b into table Branches
        end for
    end for
end procedure
```

Providing user recommendations. Users only have a limited interest in evolution. They do not care about files on other development lines because they are out of their reach. Therefore, only files on the current branch are important to make recommendations to programmers. Deleted files should not be considered because they cannot be changed anymore. A very simple approach to get all existing files on a branch is to checkout the branch.

Of course, it is possible and even practical to extract everything, and then restrict the relevant files according to the later application.

3.3 Group Checkins to Transactions

Most modern version control systems have a concept of *product versioning*—that is, one is able to access commits that alter the entire product. However, CVS provides only *file versioning* discarding the relations between files of a commit. But this information is essential for ROSE as the mining approach is based on it. Thus, we must *group* the individual per-file changes



```
RCS file: /home/eclipse/org.eclipse.jdt.core/model/org/eclipse/jdt/core/IBuffer.java,v
Working file: ./org.eclipse.jdt.core/model/org/eclipse/jdt/core/IBuffer.java
head: 1.17
branch:
locks: strict
access list:
symbolic names:
         v_397: 1.16
         v_396a: 1.16
         ...
         v_382: 1.15
         JDK_1_5: 1.15.0.2
         Root_JDK_1_5: 1.15
         v_381: 1.15
         ...
keyword substitution: o
total revisions: 24;selected revisions: 24
description:
----------------------------
revision 1.17
date: 2004/01/13 15:48:42;  author: jlanneluc;  state: Exp;  lines: +1 -1
Updated copyrights to 2004
----------------------------
revision 1.16
date: 2003/12/15 16:25:37;  author: jlanneluc;  state: Exp;  lines: +15 -26
46040
----------------------------
revision 1.15
date: 2003/05/26 16:13:24;  author: pmulet;  state: Exp;  lines: +5 -1
branches:  1.15.2;
*** empty log message ***
----------------------------
...
----------------------------
revision 1.15.2.1
date: 2004/01/12 19:53:11;  author: othomann;  state: Exp;  lines: +15 -26
Merge with HEAD
=============================================================================
```

Figure 3.2: Extract Information from CVS *log* output

(also called checkins) into individual transactions. We distinguish between *commits* performed by developers (by calling CVS commit) and inferred *transactions* used by ROSE for mining. Ideally, one commit matches exactly one transaction, and vice versa.

There are two approaches to infer transactions from checkins: *time windows* and *commit mails*.

3.3.1 Time Windows

An obvious solution for grouping checkins is to consider all changes by the same developer, with the same log message, made at the same time as one *transaction*. But the term "same time" is inaccurate in this context because usually, commit operations take several seconds or minutes—especially if many files are involved. In practice, many approaches consider not only checkins at the same time as candidates, but also checkins during a time interval:

Fixed time windows restrict the maximal duration of a transaction. The time interval always begins at the *first checkin*. This approach has been used with a time window of three minutes by [MFH02, GJK03] for the analysis of CVS archives.

Sliding time windows restrict the maximal gap between two subsequent checkins of a transaction. The begin of the time interval is shifted to the *most recent checkin*. Thus, this approach can recognize transactions that take longer to complete than the duration of the time window. This approach originates from *ChangeLog* programs like *cvs2cl*, and is called the "Right Way" by its developers including Karl Fogel [FO02].

Figures 3.3 and 3.4 illustrate the difference between fixed and sliding time windows. Figure 3.3 uses a fixed time window: After the checkin of A:1.3, both checkins B:1.2 and C:1.4 are part of the same transaction because they are visible within the time window (drawn in white). D:1.3 and E:1.5 are outside the time window and therefore considered as a new transaction.

Figure 3.4 shows that a sliding time window additionally considers D:1.3 and E:1.5 because the time window “slides” from checkins A:1.3 to finally E:1.5 (see Figures 3.4(a)) to 3.4(e)). The transaction (light gray) is closed after E:1.5 as no further checkins are visible within the time window (drawn in white).

Same Author, Message, and Time

Formally, using a sliding time window of 200 seconds, for all checkins $\delta_1, \ldots, \delta_k$ that are part of a transaction Δ, the following conditions hold (without loss of generality δ_is are sorted by *time*(δ_i)):

$$\forall \delta_i \in \Delta : \mathit{author}(\delta_i) = \mathit{author}(\delta_1) \tag{3.1}$$

$$\forall \delta_i \in \Delta : \mathit{log\_message}(\delta_i) = \mathit{log\_message}(\delta_1) \tag{3.2}$$

$$\forall i \in \{2, \ldots, k\} : |\mathit{time}(\delta_i) - \mathit{time}(\delta_{i-1})| \leq \langle 200 \text{ sec}\rangle \tag{3.3}$$

For transactions Δ of size two or more we can rewrite Condition 3.3:

$$\forall \delta_a \in \Delta : \exists \delta_b \in \Delta, \delta_a \neq \delta_b : |\mathit{time}(\delta_a) - \mathit{time}(\delta_b)| \leq \langle 200 \text{ sec}\rangle \tag{3.4}$$

Condition 3.3 for a fixed time window is similar to 3.4—except for the $\exists$ quantifier which is now a $\forall$ quantifier:

$$\forall \delta_a \in \Delta : \forall \delta_b \in \Delta, \delta_a \neq \delta_b : |\mathit{time}(\delta_a) - \mathit{time}(\delta_b)| \leq \langle 200 \text{ sec}\rangle \tag{3.5}$$

This difference is quite straightforward: For sliding time windows only two subsequent changes have to be within 200 seconds—corresponds to $\exists$)—while for fixed time windows all changes have to be less than 200 seconds apart—corresponds to $\forall$.

There are two additional conditions for transactions (valid for both fixed and sliding time windows): *mutually exclusive files* and *no interleaving of transactions*.

Mutually Exclusive Files

Each file can only be part of a single transaction once because CVS does not allow to commit two revisions of a file at the same time. For a transaction $\Delta = (\delta_1, \ldots, \delta_n)$ this means, that all checkins δ_a and δ_b have to affect mutually exclusive files:

$$\forall \delta_a, \delta_b \in \Delta : \; \delta_a \neq \delta_b \Rightarrow \mathit{file}(\delta_a) \neq \mathit{file}(\delta_b) \tag{3.6}$$

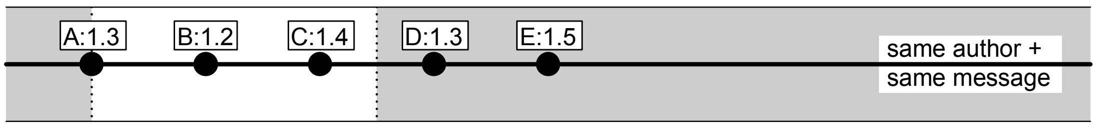
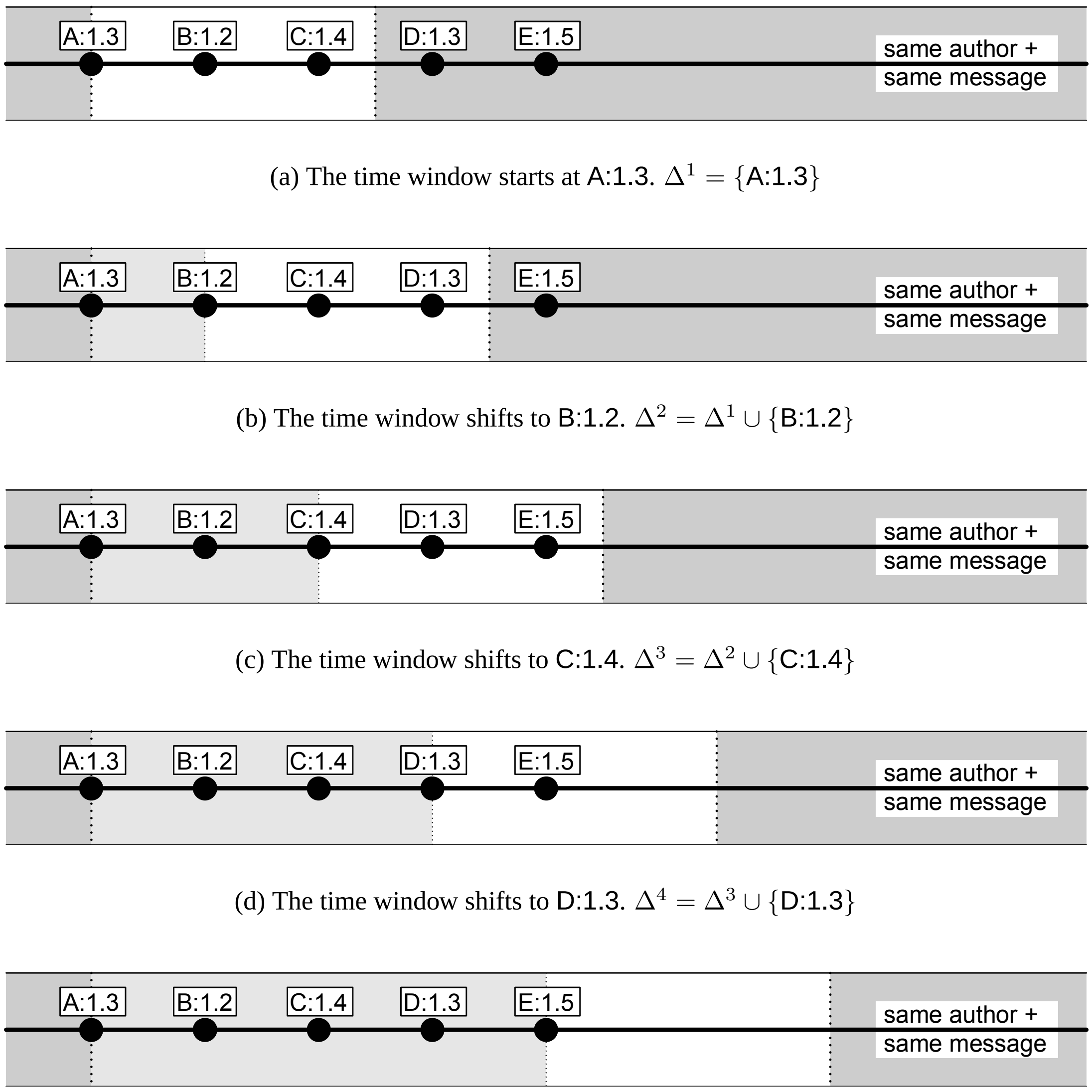


Figure 3.3: Fixed Time Windows



(a) The time window starts at A:1.3. $\Delta^1 = \{\text{A:1.3}\}$

(b) The time window shifts to B:1.2. $\Delta^2 = \Delta^1 \cup \{\text{B:1.2}\}$

(c) The time window shifts to C:1.4. $\Delta^3 = \Delta^2 \cup \{\text{C:1.4}\}$

(d) The time window shifts to D:1.3. $\Delta^4 = \Delta^3 \cup \{\text{D:1.3}\}$

(e) The time window shifts to E:1.5. $\Delta^5 = \Delta^4 \cup \{\text{E:1.5}\}$. No other checkins are visible; so, the transaction is closed. $\Delta = \Delta^5 = \{\text{A:1.3}, \text{B:1.2}, \text{C:1.4}, \text{D:1.3}, \text{E:1.5}\}$.

Figure 3.4: Sliding Time Windows

Algorithm 3.2 Infer Transactions Algorithm

Input: Database $\mathcal{D}$

Output: Database $\mathcal{D}$

```
1:  procedure GROUP
2:      $tid = \perp$
3:      $author = \perp$
4:      $message = \perp$
5:      $time = -1$
6:      $Files = \emptyset$

7:      Sort $Transactions \bowtie Checkins$ by $Author$, $CheckinTime$, $Message$
8:      for all rows $r$ of $Transactions \bowtie Checkins$ processed in sort order do
9:          if $r.Author \neq author$
                $\vee\ r.Message \neq message$
                $\vee\ |r.CheckinTime - time| > \langle 200 \text{ seconds} \rangle$
                $\vee\ r.FileID \in Files$ then
10:             $tid = r.TransactionID$                        /* Found a new transaction */
11:             $author = r.Author$
12:             $message = r.Message$
13:             $Files = \emptyset$
14:         else                                   /* Assign checkin to correct transaction */
15:             Update table $Checkins$ set new $tid$ for $r.FileID$, $r.RevisionID$
16:         end if
17:         $time = r.CheckinTime$                         /* We have a sliding time window */
18:         $Files = Files \cup \{r.FileID\}$
19:     end for
20:     Remove all unreferenced transactions in table $Transactions$
21: end procedure
```

No Interleaving of Transactions

A developer cannot perform two different transactions at the same time. In other words, if she begins a new transaction, all her previous transactions have to be completed. This means that for a transaction Δ the following must hold:

$$\forall \delta \in \Delta :\ begin\_time(\Delta) \leq time(\delta) \leq end\_time(\Delta) \wedge author(\delta) = author(\Delta) \\ \Rightarrow log\_message(\delta) = log\_message(\Delta) \quad (3.7)$$

This condition is difficult to realize because in some cases CVS inserts dummy checkins, e.g., if a file was initially added on a branch and merged into another branch later on. Such checkins have to be treated separately to avoid incidentally breaking up a merge into several transactions.

tid	FileName	RevisionID	Author	CheckinTime	Message	
#1	A	1.3	Frodo	2004-01-23 10:23:17 p.m.	L1	
	B	1.2	Frodo	2004-01-23 10:24:11 p.m.	L1	
#2	C	1.4	Frodo	**2004-01-24 07:41:27 a.m.**	**L2**	▷1
	D	1.7	Frodo	2004-01-24 07:43:33 a.m.	L2	
#3	**C**	1.3.1.4	Frodo	2004-01-24 07:45:07 a.m.	L2	▷2
#4	E	1.5	Frodo	**2004-01-24 03:52:07 p.m.**	**L3**	▷3
#5	F	1.3	Frodo	2004-01-24 03:54:17 p.m.	**L4**	▷4
#6	G	1.9	Frodo	2004-01-24 03:55:00 p.m.	**L3**	▷5
#7	H	1.8	**Gandalf**	**2004-01-22 08:14:23 p.m.**	**L5**	▷6

	Reasons for new transactions:
▷1	message is different; interval to previous checkin exceeds 200 seconds
▷2	file C is already in transaction #2
▷3	message is different; interval to previous checkin exceeds 200 seconds
▷4	message is different
▷5	message is different
▷6	author and message are different; interval to previous checkin exceeds 200 seconds

Figure 3.5: Grouping Revisions to Transactions

The Algorithm for Inferring Transactions

The algorithm for grouping checkins to transactions is straightforward (see Algorithm 3.2 on the preceding page): Simply sort checkins by author, checkin time, and log message. Iterate over checkins in this order: Each time the author or log message differs from the ones of the previous checkin, or the time window is exceeded, start a new transaction. Sorting by author, log message and checkin time is also possible, but ignores Condition 3.7 (and allows interleaving of transactions). An example application of the grouping algorithm is illustrated in Figure 3.5. Differing attributes that result in new transactions are in boldface.

Based on our experience, sliding time windows are superior to fixed time windows because they deal with transactions of any duration. The selection of the length of a time window (fixed or sliding) depends on the analyzed project and the analysis itself. The time window should be chosen based on the assumption on how long it takes to check in the largest file with high network latency. Up to now, most lengths of time windows are arbitrary: They range from two to four minutes. The length of time windows is discussed in detail in Section 3.3.3.

3.3.2 Commit Mails

Time windows are a good approximation for inferring transactions from CVS. A more precise solution is based on *commit mails*—that are mails sent to developers after a commit. The example in Figure 3.6 on the next page shows, that such a mail contains the committer, the timestamp, the modified files, and the log message. With this information, it is straightforward to relate files to revisions, and then to commits.

```
CVSROOT: /cvs/gcc
Module name: gcc
Changes by: zack@gcc.gnu.org 2004-05-01 19:12:47

Modified files:
gcc/cp         : ChangeLog decl.c

Log message:
* decl.c (reshape_init): Do not apply TYPE_DOMAIN to a VECTOR_TYPE.
Instead, dig into the representation type to find the array bound.

Patches:
http://.../cvsweb.cgi/gcc/gcc/cp/ChangeLog.diff?...&r2=1.4042
http://.../cvsweb.cgi/gcc/gcc/cp/decl.c.diff?...&r2=1.1204
```

Figure 3.6: Commit Mail

Unfortunately, the solution based on commit mails has two major drawbacks:

1. *Suitable commit mails are only available for few projects.*
   Many projects (especially projects hosted at Sourceforge.net) send commit mails for each directory separately, which makes it hard to restore transactions. In practice, one has to use time windows to deal with such mails.

2. *Commit mails and CVS data are difficult to integrate.*
   All data stored in CVS is dynamic which means, that it can be changed at any time; even log messages can be modified. In contrast, commit mails are static—once sent, they remain unchanged. Thus, CVS archives and commit mails diverge during time, and it is tricky to bring them back together.

Therefore, commit mails are of limited use for restoring commits. However they are useful to adjust the length of time windows as we show in the next section.

3.3.3 Choosing the Time Window Length

We restored all commits of GCC between 2000-06-01 and 2003-06-01. Using the commit mail approach we inferred a total of 32,529 commits. We will use these actual commits to determine lower and upper bounds for the length of time windows.

How long are commits?

The duration of a commit Δ is the difference between the timestamps of the first and the last checkin:

$$\mathit{duration}(\Delta) = \mathit{end\_date}(\Delta) - \mathit{begin\_date}(\Delta)$$

Table 3.1 on the following page shows that although the average duration is around six seconds, there are commits that take more than 21 minutes. This confirms the proposition that fixed time windows are not reasonable because the duration of commits is unbounded. (Recall that most fixed time window approaches use a time window of three minutes.)

Within one commit, what is the maximal distance between two subsequent checkins?

Two checkins δ_a and δ_b are *subsequent* within one commit Δ if no other checkin exists between them (without loss of generality $\mathit{time}(\delta_a) \leq \mathit{time}(\delta_b)$):

$$\forall \delta_x \in \Delta, \delta_x \neq \delta_a, \delta_x \neq \delta_b : \mathit{time}(\delta_x) \leq \mathit{time}(\delta_a) \vee \mathit{time}(\delta_b) \leq \mathit{time}(\delta_x)$$

The *distance* between two subsequent checkins δ_a and δ_b is defined as:

$$\mathit{distance}(\delta_a, \delta_b) = |\mathit{time}(\delta_a) - \mathit{time}(\delta_b)|$$

Measuring the distance between two subsequent checkins of the same commit gives us a lower bound for the length of a sliding time window. This distance varies for different files and is influenced by:

- *Number of Revisions & Size of RCS Files.*
  For an ASCII file, CVS only stores the differences between two revisions in a corresponding RCS file. For a commit, the latest revision is needed and assembled on demand by applying all existing differences. The speed of this assembly depends on the number of revisions, and the size of the difference (which is the size of the RCS file).

- *Size of Files.*
  Additionally, the size of a file has impact on the distance between two checkins because the content of a file is transferred over network before the commit operation takes place.

Table 3.2 on the next page shows the highest measured distances per file. Two files stick out: gcc/libstdc++-v3/configure with 10 minutes 28 seconds, and gcc/gcc/ChangeLog with 7 minutes 12 seconds. This suggests a sliding time window length of at least 10 minutes 28 seconds.

Additionally, Table 3.2 compares the maximal distances to the number of revisions, the size of the corresponding RCS file, and the size of the file itself. Values that belong to the ten highest values in a category are in italics.

Note that it does not matter if entries of Table 3.2 are generated from other files. In most cases, they are generated by developers and not by CVS. Even if CVS would create them automatically, we have to consider them because otherwise one commit could be split incidentally into two commits.

Author	Log Message	# Files	Duration
matz	merged with ra-merge-initial	5,910	21 min 17 sec
dnovillo	Merge with mainline as of 2002-03-04.	1,087	21 min 03 sec
dnovillo	Mainline merge as of 2002-05-26.	596	18 min 03 sec
matz	merge in head from ra-merge-20020521	791	17 min 56 sec
geoffk	Merge to tag pch-merge-20020430. *[. . .]*	2,672	15 min 15 sec
⋮	⋮	⋮	⋮
		∅ 8.78	∅ 6 sec

Table 3.1: Duration of Commits

File Name	Max. Distance	# Rev.	Size of RCS	File Size
gcc/libstdc++-v3/configure	**10 min 28 sec**	550	*33,570 KB*	*715 KB*
	5 min 17 sec			
	5 min 13 sec			
	4 min 44 sec			
	2 min 53 sec			
gcc/gcc/ChangeLog	**7 min 12 sec**	*22,097*	*32,515 KB*	600 KB
	3 min 41 sec			
	3 min 15 sec			
	2 min 39 sec			
	2 min 36 sec			
gcc/gcc/po/gcc.pot	2 min 15 sec	53	*8,662 KB*	450 KB
gcc/libstdc++-v3/config.h.in	2 min 05 sec	150	211 KB	25 KB
gcc/libstdc++-v3/acconfig.h	1 min 23 sec	69	114 KB	10 KB
gcc/libstdc++-v3/acinclude.m4	1 min 23 sec	424	843 KB	80 KB
gcc/libstdc++-v3/aclocal.m4	1 min 23 sec	446	1,306 KB	87 KB
gcc/libstdc++-v3/ChangeLog	1 min 21 sec	*2,522*	4,683 KB	153 KB
gcc/libstdc++-v3/configure.in	1 min 21 sec	272	312 KB	18 KB
gcc/gcc/po/fr.po	1 min 17 sec	34	*10,645 KB*	*818 KB*
gcc/libstdc++-v3/Makefile.in	1 min 17 sec	243	315 KB	173 KB
gcc/gcc/po/es.po	1 min 12 sec	29	*9,623 KB*	*820 KB*
gcc/gcc/cp/ChangeLog	1 min 08 sec	*4,392*	*8,057 KB*	660 KB
gcc/libstdc++-v3/Makefile.am	58 sec	112	98 KB	5 KB
⋮	⋮	⋮	⋮	⋮
(further 21,831 files)	⋮	⋮	⋮	⋮
		∅ 17.4	∅ 41 KB	∅ 6 KB

Table 3.2: Maximal Distance between two Subsequent Checkins within one Commit

What is the minimal distance between two similar commits?

Two commits Δ_1 and Δ_2 are *similar* if they have the same developer, the same log messages, and all files are mutually disjoint.

$$\begin{gathered} \mathit{author}(\Delta_1) = \mathit{author}(\Delta_2) \\ \mathit{log\_message}(\Delta_1) = \mathit{log\_message}(\Delta_2) \\ \forall \delta_a, \delta_b \in \Delta_1 \cup \Delta_2 : \ \delta_a \neq \delta_b \Rightarrow \mathit{file}(\delta_a) \neq \mathit{file}(\delta_b) \end{gathered}$$

In other words, without Condition 3.4 (for same time) but all other conditions valid (same author, same message, no interleaving, and *especially* mutually exclusive files), they would be considered as one single transaction. Measuring the minimal distance between two similar commits Δ_1 and Δ_2 (without loss of generality $\mathit{begin\_time}(\Delta_1) \leq \mathit{begin\_time}(\Delta_2)$) allows us to determine an upper bound for the length of sliding time windows:

$$\mathit{distance}(\Delta_1, \Delta_2) = \mathit{max}\,(0, \mathit{begin\_time}(\Delta_2) - \mathit{end\_time}(\Delta_1))$$

Table 3.3 on the following page shows the results for GCC. Surprisingly, the minimal distance is only *one* second. This means that, if we use time windows it is not possible to infer transactions that match exactly one commit operation. In practice, a transaction consists of several commits.

Two log messages deserve some explanation: "Mark ChangeLog" appears more than 472 times, by two authors and on 14 different dates. Each of these commits inserts the release of a new GCC version into one of the numerous ChangLog files. "Update version" does the same for several version files (e.g., version.c). Again, for each of these files one commit is performed.

Often developers consciously call a commit operation several times with the same log message. In this case, for each call one commit mail is sent. One example is the log message "PR java/10145…" with a total of only two commits. As they are only 7 seconds apart, it is very unlikely that they are unrelated. This is another disadvantage of the commit mail approach: One logical change can be distributed over several commits. To deal with such situations, we would have to introduce time windows again.

Note that some commits are performed daily, like "Bump dates." or "Daily bump." Both commits insert the current date into the version files of GCC. These commits do not match the definition of similar commits because they all affect the same files (Condition 3.6 for mutually exclusive files is violated). Thus, they do not manifest in Table 3.3.

What are the consequences for ROSE?

The above results indicate that sliding time windows are superior to fixed time windows.

The average distance between checkins for the two outliers of Table 3.2 is 23 seconds for gcc/libstdc++-v3/configure, and 7 seconds for gcc/gcc/ChangeLog. In addition, for only a few checkins this distance exceeds three minutes. Based on these observations, we take three minutes as the lower bound for the length of time windows.

Author	Log Message	Min. Distance
aoliva	* configure.in: Propagate ORIGINAL_LD_FOR_MULTILIBS to config.status. * configure: Rebuilt.	1 sec
gdr	Mark ChangeLog	2 sec
mmitchel	* decl.c (grokfndecl): Require that ‘main’ return an ‘int’. * mangle.c (write_encoding): Don’t mangle return types for conversion functions.	2 sec
mmitchel	Mark ChangeLog	2 sec
mmitchel	Update version	2 sec
gdr	Update version	3 sec
mmitchel	* expr.c (expand_expr, case ARRAY_REF): Correct check for side-effects in the value of an array element.	5 sec
jason	PR java/10145 * stor-layout.c (update_alignment_for_field): Respect DECL_USER_ALIGN for zero-length bitfields, too. * c-decl.c (finish_struct): Don’t set DECL_ALIGN for normal fields. * cp/class.c (check_field_decl): Don’t set DECL_ALIGN.	7 sec
⋮	⋮	⋮
⋮	(further 705 log message)	<5 min 00 sec
⋮	⋮	⋮
bkoz	Benjamin Kosnik <bkoz@fillmore.redhat.com> * acinclude.m4: Change up to reflect new directory organization. Add in bits for NetBSD. * aclocal.m4: Regenerate. * configure: Regenerate. * config/os/bsd: New directory.	5 min 03 sec
⋮	⋮	⋮
⋮	(further 70 log message)	⋮
⋮	⋮	⋮
cgf	Merge from 3.2.1.	36 min 35 sec
jason	PR c++/7279 * tree.c (cp_copy_res_decl_for_inlining): Also copy TREE_ADDRESSABLE.	42 min 41 sec
bkoz	2001-04-02 Stephen M. Webb <stephen@bregmasoft.com> * include/c_std/bits/std_cstring.h: Fix for const-correctness. * include/c_std/bits/std_cwchar.h: Same. * testsuite/21_strings/c_strings.cc: Add.	49 min 32 sec

Table 3.3: Minimal Distance for Similar Commits

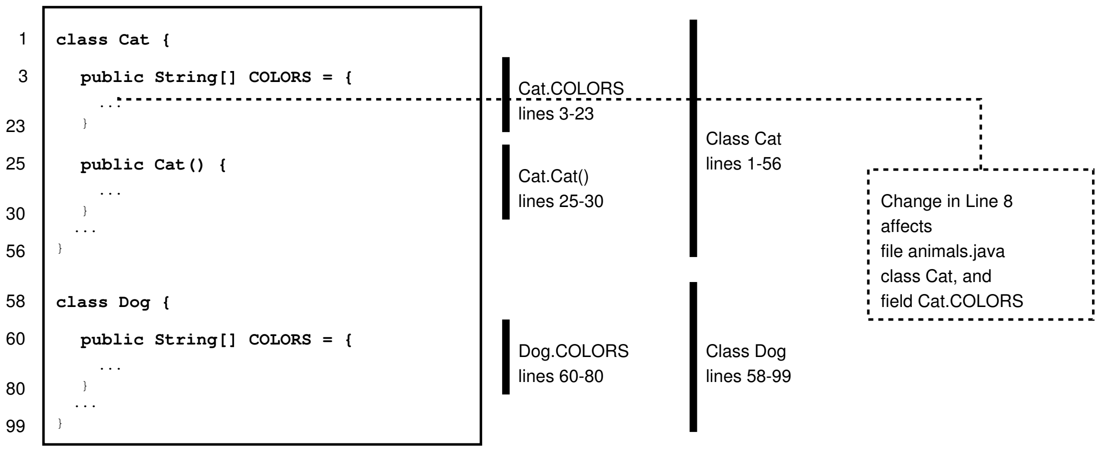


Figure 3.7: Changes on Lines Affect Entities

Using 49 minutes as an upper bound is not reasonable because most developers are fast commiters. About 90% of all log messages in Table 3.3 have a distance of less than five minutes. Thus, we consider five minutes as an upper bound.

To summarize: The time window should be between three and five minutes. Using such a time window, it is not possible to exactly approximate commits with transactions. However, only related commits are composed to one transaction.

ROSE uses a sliding time window of 200 seconds, which is three minutes plus a *buffer* of 20 seconds. Without this buffer, the end of the time window can clash with the release of a CVS *lock*. In this case, the continuation of an interrupted transaction would be considered as a new transaction.

3.4 Map Changes to Entities

CVS provides only information on changed files but not on changed functions. Thus, another preprocessing step is required: Each revision is compared with its predecessor, and the changes are mapped to syntactic components of files. Revisions with no predecessors are compared against an empty file. Figure 3.7 sketches the idea that a changed line affects at least one entity, e.g., line 8 affects the field Cat.COLORS, in class Cat, in file Animals.java.

Without this preprocessing step, ROSE could only make recommendations on file-level, and would miss many interesting relations, thus being only of limited use.

3.4.1 The Framework

Fine-grained changes can be computed using a *diff*-tool and a light-weight analysis that creates the building blocks of files. This approach is open to everything: source code, documentation, XML files, and even diagrams or images. For a change from revision r_1 to r_2 we compute the entities as follows:

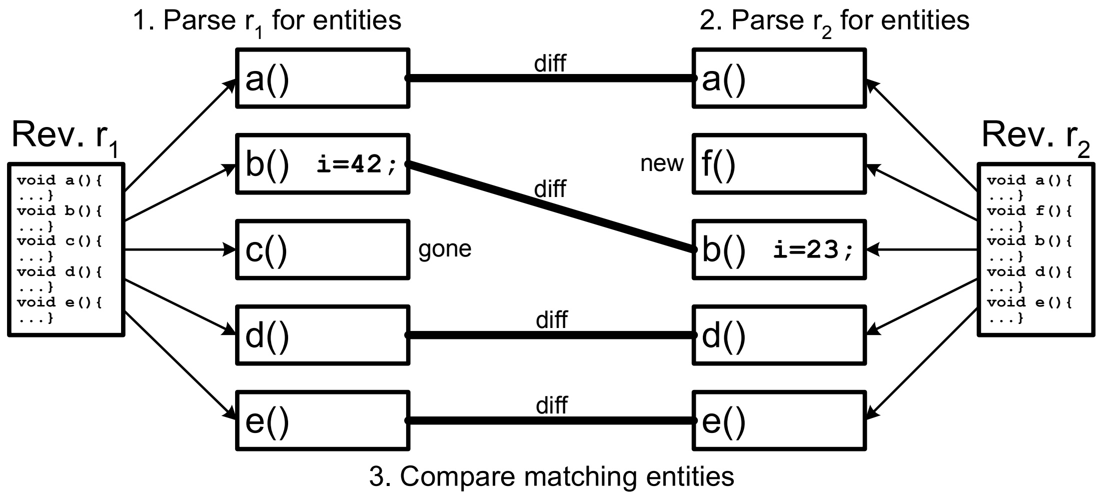

1. Parse r1 for entities
2. Parse r2 for entities
Rev. r1
Rev. r2
void a(){ ...} void b(){ ...} void c(){ ...} void d(){ ...} void e(){ ...}
void a(){ ...} void f(){ ...} void b(){ ...} void d(){ ...} void e(){ ...}
a()
b() i=42;
c()
d()
e()
a()
f()
b() i=23;
d()
e()
diff
new
diff
gone
diff
diff
3. Compare matching entities

Figure 3.8: Map Changes to Entities

1. Determine all entities E_1 of revision r_1 and all entities E_2 of revision r_2.
2. The *added* entities are $E_2 - E_1$, and the *removed* entities are $E_1 - E_2$.
3. All entities in $E_1 \cap E_2$ *may* have been changed. Whether an entity e has actually been changed is decided by performing a *diff* between the source-code of e in r_1 and its source-code in r_2. The set of all changed entities is C.

Figure 3.8 outlines the above algorithm. First the sets of entities are determined: $E_1 = \{\text{a()}, \text{b()}, \text{c()}, \text{d()}, \text{e()}\}$ for revision r_1 and $E_2 = \{\text{a()}, \text{f()}, \text{b()}, \text{d()}, \text{e()}\}$ for r_2. Until now, we know that in revision r_2 function c() has been removed ($E_1 - E_2$) and f() has been inserted ($E_2 - E_1$). Next we compare for each entity the respective source code parts (indicated by thick lines) and recognize that b() has actually been changed. All other entities have been unchanged because the *diff* between them is empty.

If an entity e is renamed to f, the approach above will recognize e as deleted and f as inserted. But detecting renaming is important because otherwise all information prior to renaming is lost for mining. Using text similarity measures we can recognize such renaming: Measure the similarity of the content of each entity $e \in E_1 - E_2$ to the contents of each entity $f \in E_2 - E_1$. If a given threshold is reached then e has been renamed to f. As changes between two revisions are usually small there is no need for a sophisticated clone detecting algorithm.

ROSE provides an own extendible framework for mapping changes to syntactic entities based on the above algorithm. Extensions are provided by classes that decompose files into entities. Currently, ROSE can deal with the following file types:

JAVA, C, C++. *Classes, methods,* and *initializations* of arrays are recognized for these languages. Although the used technique sounds simple (counting brackets), dealing with preprocessor commands is tricky and involves many special cases (e.g., for conditionals). Running the preprocessor instead is not reasonable because macro expansion may hide changes.

PYTHON. Based on indentation, we recognize *classes* and *functions*. The approach is straightforward, except that tab characters have to be handled correctly. (The length of a tabulator depends on its position in a line.)

TEX, TEXINFO. *Chapters*, *sections*, and *subsections* are recognized for TEX files.

ROSE also deals with hierarchical structures. For instance, a change in a subsection also counts as a change in the enclosing chapter; or, as in Figure 3.7 the change of Cat.COLORS also affects class Cat and of course file Animals.java. Future versions of ROSE will integrate the CTAGS tool that maps tags (or entities) to source code, and vice versa.

3.4.2 Detecting Fine-Grained Changes in ECLIPSE

The ECLIPSE integration of ROSE makes use of the ECLIPSE platform [Obj03] which provides a powerful and extensible framework for comparing files:

- *Range Differencer*—The RangeDifferencer class compares two versions based on *tokens*. This approach is based on the traditional *diff* algorithm [MM85]. The tokens are created using classes that implement the interface ITokenComparator, e.g., for lines the class DocLineComparator. The calculated differences are returned in a list.

- *Structure Merge Viewer*—The Differencer class compares two versions of any given *hierarchical structure* and returns a delta tree describing each change in detail. The structure is created with an implementation of the interface IStructureCreator. The fearless like ROSE use existing *internal* classes[2], e.g., the JavaStructureCreator.

Furthermore, ECLIPSE provides an easy access to JAVA abstract syntax trees and facilitates further analysis of source code. The only drawback is that many of those features cannot be executed from the command line. For that reason, ROSE does not use ECLIPSE features during preprocessing.

3.4.3 The Database Schema for Entities

All computed fine- and coarse-grained entities are stored in the ROSE database. Figure 3.9 on the next page shows the tables that are relevant for entities:

Entities contains all entities exactly once, regardless whether they have been changed or not. An entity with name *EntityName* is identified by its *EntityID*. Additionally, it has a reference to the enclosing file *FileID*, which is a foreign key to table *Files*. The type of such an entity is stored as *EntityTypeID*, which is a foreign key to table *EntityTypes*.

[2]It is dangerous to use *internal* classes, because they may change without prior announcement [Riv01]. However, most interesting features of ECLIPSE are internal.

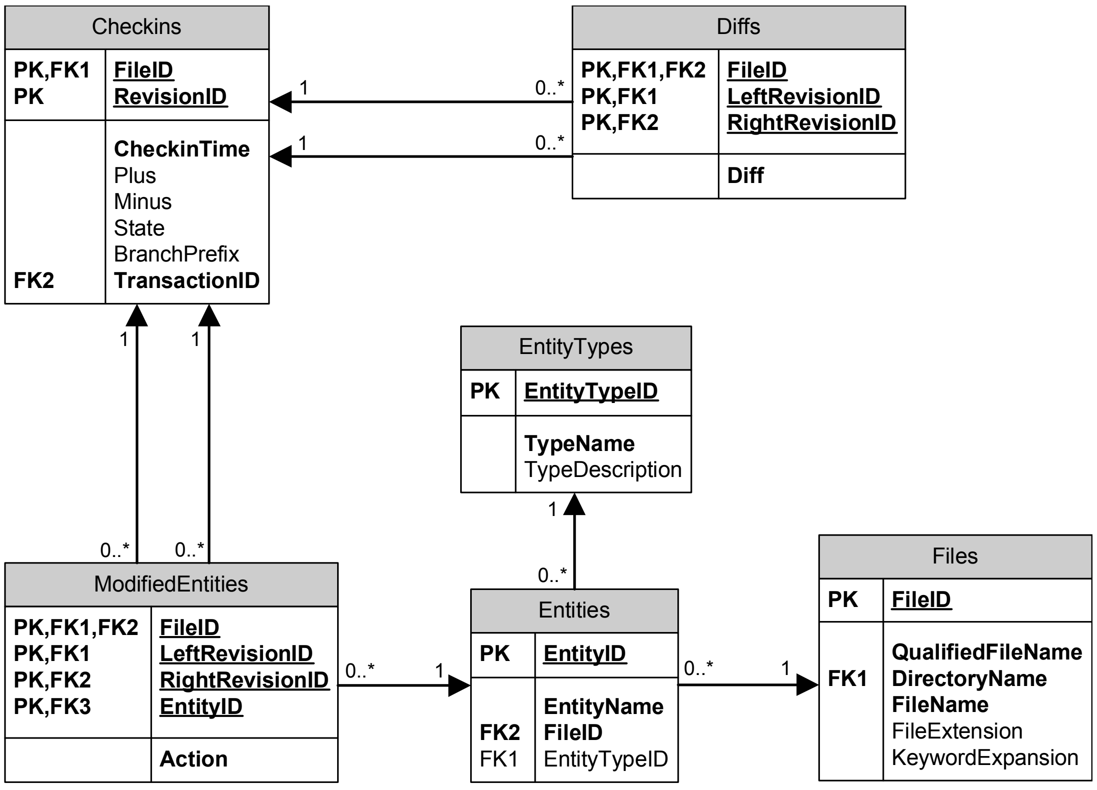


Figure 3.9: ROSE Database Schema: Tables for Entities

EntityTypes are, for instance, *files*, *classes*, *methods*, *fields*, *sections*, *subsections*, etc. A type called *TypeName* is identified by an *EntityTypeID* and described by a *TypeDescription*.

ModifiedEntities stores all entities that have been modified between two subsequent checkins. The kind of modification—*added*, *removed*, or *changed*—is stored in *Action*. Recall that *Action* and *EntityID* form an item. The first revision is referenced by (*FileID, LeftRevisionID*) and the second by (*FileID, RightRevisionID*).

Diffs contains output of the *diff* algorithm between two revisions *LeftRevisionID* and *RightRevisionID* of a file *FileID*. This output is stored in *Diff*.

Table *ModifiedEntities* contains even those entities that are not used by mining. For efficiency, a special table called *Lineitems* will be created for mining, containing only relevant items (for details see Section 3.6).

3.5 Data Cleaning

This section discusses *noise*—that is transactions that will likely induce or contribute to incorrect results—and presents appropriate cleaning techniques. However, noise evolves from several kinds of transactions: *Large transactions* often originate from infrastructure changes, *import transactions* contain complete subsystems, and *merge transactions* simply reproduce changes. Note that transactions can fit in multiple categories, e.g., large merge transactions.

ROSE performs two kinds of data cleaning:

Explicit data cleaning identifies noisy transactions *before* mining (during preprocessing) and tags them so that they can be ignored later on. (Recall the attribute *IsNoise* in table *Transactions*.)

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

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

3.5.1 Large Transactions

Large transactions are very frequent in real-life. Here are two examples from OPENSSL:

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

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

A solution is to ignore transactions of size greater than N in the analysis. The bound N depends on the examined software project. If desired, suspect transactions can be investigated manually in order to guarantee that they are actually noise.

ROSE uses $N = 30$. This bound has been determined by investigating several GCC transactions. This bound may sound low, however, a transaction of 30 files contributes to at least 2^{30} association rules and increases the complexity for traditional mining algorithms dramatically.

3.5.2 Import Transactions

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

- "Initial import of libgcj" *(371 files)*

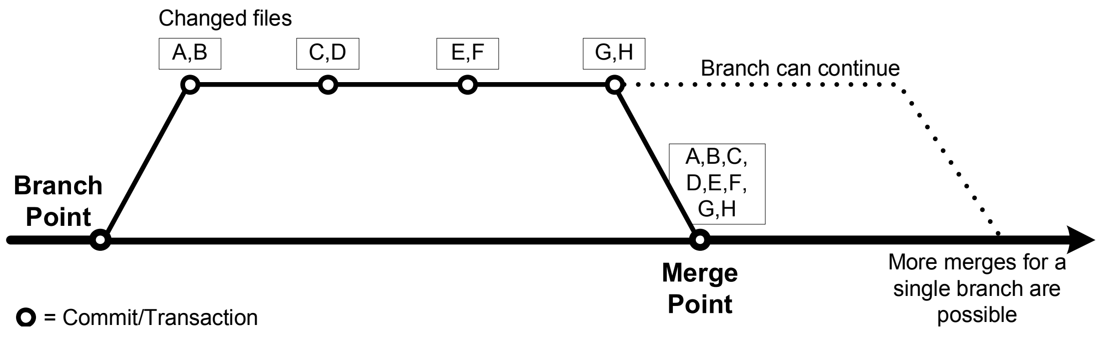


Figure 3.10: Merges Considered Harmful

- “initial import of Java front-end” *(43 files)*

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

3.5.3 Merge Transactions

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

> “mainline merge as of 2003-05-04” *(5874 files)*.

Figure 3.10 shows a smaller example. On the branch four transactions have been committed: $\{A, B\}$, $\{C, D\}$, $\{E, F\}$, and $\{G, H\}$. These files are now again changed at the merge point within a transaction that contains all changes made on the branch: $\{A, B, C, D, E, F, G, H\}$.

Merge transactions are noise for two reasons:

1. They contain unrelated changes, e.g., B and C.
2. They rank changes on branches higher (those changes are duplicated), e.g., A and B.

Taking such transactions into account has a significant influence on the results. Thus, transactions that resulted from merges should be identified and ignored.

Unfortunately, CVS does not keep track of which revisions resulted from a merge. Michael Fischer et al. proposed a heuristic to detect these revisions [FPG03b]. For each file on a branch a potential merge point is determined and verified using similarity measures. If the potential merge point is rejected, another one is tested until a valid merge point is found, or all revisions have been tested. This approach is restricted to merges to the main branch, but it is straightforward to apply it to other branches. Additionally, they work only on revisions instead of analyzing complete transactions. Analyzing transactions simplifies the detection of merges because if a merge is detected for a single file, the whole transaction is probably a merge. Nonetheless, automatic merge detection is difficult to realize because of the large number of existing merge policies. For example, as Figure 3.10 indicates, the development can continue on both branches after a merge, creating additional complexity for all heuristics.

A simple but powerful *suspect & verify* approach is based on log messages and the observation that merges are well-documented in those messages:

1. All transactions whose log message contains a “merge” (case-insensitive) are *suspect* to be a merge transaction.
2. Check each suspect transaction manually and *verify* merge or not. This step is essential to avoid errors for log messages that incidentally contain a “merge”, like the ECLIPSE transaction “New isMerge(), isMergeWithConflicts(), and setMerge() methods”.

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

3.6 The Output of Preprocessing

The output of the preprocessing phase are fine-grained changes, represented by items and grouped to transactions. All these results are linked in one database table (see Figure 3.11 on the next page).

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

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

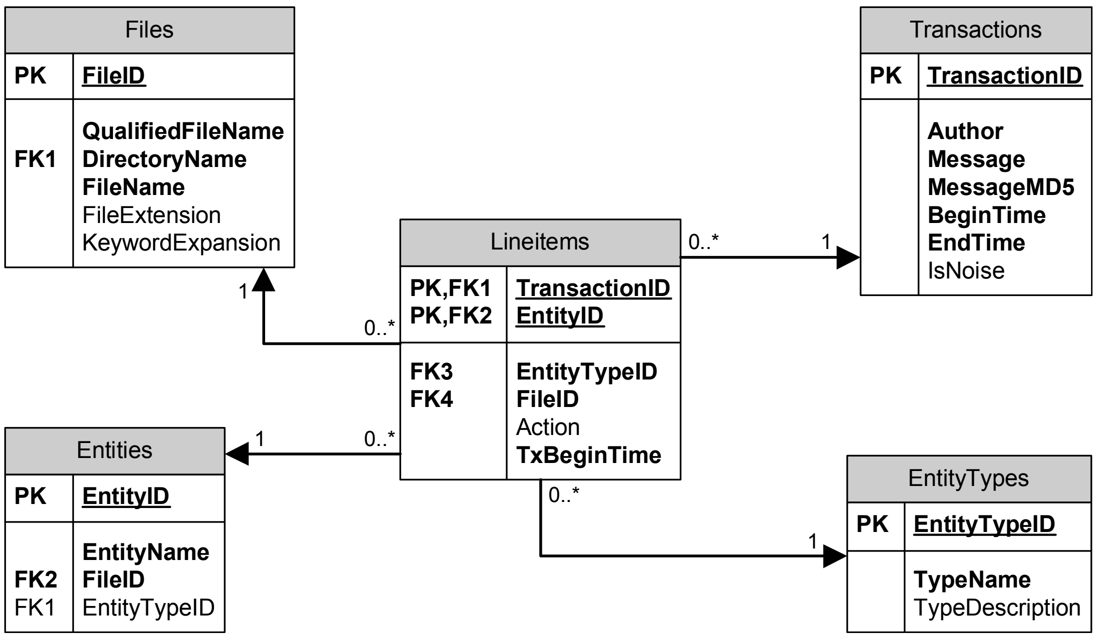


Figure 3.11: ROSE Database Schema: Tables for Mining

Chapter 4

Mining Association Rules

There are far more papers published on algorithms to discover association rules than there are papers published on applications of association rules.
– David Hand, Heikki Mannila, Padhraic Smyth in [HMS01]

The most popular application for association rule mining is market basket analysis. Based on sales transactions, frequent patterns are searched and returned as association rules. One common example is that diapers and beer often are sold together. Such information is valuable for cross-selling, thus increasing the total sales of a company. For instance, a supermarket can place beer next to diapers hinting to parents that they should buy not only necessities for their baby but also luxury for themselves.

ROSE does pretty the same thing for software developers. It searches for patterns within the version history and presents related entities in a view next to the source code (recall Figure 1.2 on page 3). However, we have to keep in mind that the objectives of ROSE are different from those of supermarkets: ROSE does not benefit by selling diapers, beer, or changes. In other words, it has no interest in increasing the total number of changes by developers.

The objectives pursued by ROSE are:

- Simplify navigation through source code.
- Avoid errors due to missing updates.

We can think of ROSE as the owner of a small shop around the corner whose main interest are happy customers. Of course, ROSE has to take a small fee which is the version history plus some computing power.

In this chapter we will discuss the techniques used by ROSE to provide its functionality. We will start with the concept of association rules and show how ROSE uses them. Afterwards, we will introduce a general mining technique, called the Apriori algorithm, and present an improved mining algorithm for ROSE.

4.1 Association Rules

As indicated before association rules represent some *pattern*. For instance, the following rule represents the pattern {fKeys[], initDefaults(), plugin.properties} within the version history of ECLIPSE:

$$\text{change(fKeys[])} \Rightarrow \text{change(initDefaults())} \wedge \text{change(plugin.properties)} \quad [0.875]$$

There are two different interpretations for this rule:

- The *descriptive* interpretation directs to the past: Whenever the user changed the field fKeys[], she also changed the method initDefaults() and the file plugin.properties with a certainty of 87.5%.
- In contrast, *predictive* interpretation (as used by ROSE) directs to the future: Now, the rule means that, whenever the user changes the field fKeys[], she *should* also change the method initDefaults() and the file plugin.properties. Here, "should" means that the rule is based on experience (with a certainty of 87.5%) and does not constitute absolute truth[1]; the character "⇒" is thus not to be read as a logical implication that always holds.

As mentioned before, rules have a *probabilistic* interpretation based on the *amount of evidence* in the transactions they are derived from. This amount of evidence is determined by three measures:

Frequency. The *frequency* (or *count*) 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 plugin.properties. Then, the frequency for the above rule is 7.

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

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

For the formal definition of association rules we adopt [HMS01]. Let $\mathcal{I} = \{i_1, \ldots, i_n\}$ be the set of all items recognized during preprocessing. Recall that an *item* is an entity e combined with an action[2], e.g., change entity e. Let $\mathcal{D}$ be the task-relevant data, i.e., the set of all transactions[3], where each transaction T is a set of items such that $T \subseteq \mathcal{I}$.

[1] Note that using predictive interpretation an association rule *never* can constitute absolute truth, even if its certainty is 100%.

[2] Currently, ROSE mines only items where the action is "change". The resulting rules are called *single-dimension association rules* because each action corresponds to one dimension.

[3] Recall that we have two different notations for transactions: Δ is based on the composition of changes represented by functions, and T is a list of items, in particular of the changed entities. ROSE uses the latter for mining.

An *association rule* is an implication of the form $A \Rightarrow B$ where $A \subset \mathcal{I}$, $B \subset \mathcal{I}$, and $A \cap B = \emptyset$. The *antecedent* of a rule is A and the *consequent* is B. Usually, A and B are conjunctions, but it is possible to use any kind of proposition. However, we will focus on conjunctions.

Formally, we define the frequency, support, and confidence of an association rule as follows:

- The *frequency* of a set X in the task-relevant data $\mathcal{D}$ is defined as:

  $$\textit{frequency}_{\mathcal{D}}(X) = |\{T | T \in \mathcal{D}, X \subseteq T\}|$$

  The *frequency* of an association rule $A \Rightarrow B$ in the task-relevant data $\mathcal{D}$ is defined as:

  $$\textit{frequency}_{\mathcal{D}}(A \Rightarrow B) = \textit{frequency}(A \cup B)$$

- The *support* of a set X in the task-relevant data $\mathcal{D}$ is defined as:

  $$\textit{support}_{\mathcal{D}}(X) = \frac{\textit{frequency}_{\mathcal{D}}(X)}{|\mathcal{D}|} = P(X)$$

  The *support* of an association rule $A \Rightarrow B$ in the task-relevant data $\mathcal{D}$ is defined as:

  $$\textit{support}_{\mathcal{D}}(A \Rightarrow B) = \frac{\textit{frequency}_{\mathcal{D}}(A \cup B)}{|\mathcal{D}|} = P(A \cup B)$$

- The *confidence* of an association rule $A \Rightarrow B$ in the task-relevant data $\mathcal{D}$ is defined as:

  $$\textit{confidence}_{\mathcal{D}}(A \Rightarrow B) = \frac{\textit{frequency}_{\mathcal{D}}(A \cup B)}{\textit{frequency}_{\mathcal{D}}(A)} = P(B|A)$$

We omit the task-relevant data $\mathcal{D}$ if it is known in the context or irrelevant. The shorthand notation $r[s; c]$ denotes a rule r with $s = \textit{support}(r)$ and $c = \textit{confidence}(r)$.

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

Obviously, the support threshold can be replaced by a frequency threshold:

$$\textit{min\_freq} = \lceil \textit{min\_supp} \cdot |\mathcal{D}| \rceil$$

ROSE uses frequency instead of support for two reasons:

1. *Frequency is easier to understand for developers.*
   Support values like 0.00016 give them no idea whether this value is high or low. In contrast, the corresponding frequency of 7 clearly expresses the significance of the rule.

2. *Frequency allows comparison of different projects.*
   It is not possible to reuse a support threshold for another project. Consider ECLIPSE with a total of 44,786 transactions, and JEDIT with only 1,905 transactions. Using $\textit{min\_supp} = 0.0001$ we mine in ECLIPSE with $\textit{min\_freq} = 5$, but in JEDIT only with $\textit{min\_freq} = 1$. Using such low frequency thresholds is not reasonable.

Confidence is an Indicator for Certainty.

The confidence of a rule can be misleading as the example below shows:

	B	$\overline{B}$	
A	20%	5%	25%
$\overline{A}$	70%	5%	75%
	90%	10%	100%

Consider the rule $A \Rightarrow B$ with the high confidence of $P(B \mid A) = \frac{P(A \cap B)}{P(A)} = \frac{0.20}{0.25} = 0.80$. This rule is misleading since the rule has a high confidence, but the occurrence of A actually *decreases* the likelihood of B from $P(B) = 0.90$ to $P(B \mid A) = 0.80$. This observation led to a variant of association rule mining called correlation mining [SBM98].

ROSE does not care if a rule is misleading or not. In the example above, B has been changed in 80% after a change on A, and is therefore still important for developers.

Support is an Indicator for Statistical Significance.

The support value of a rule is a measure for its statistical significance. If the following condition holds, A and B are likely independent and their co-occurrence in transactions is incidentally:

$$\textit{support}(A \Rightarrow B) \approx \textit{support}(A) \cdot \textit{support}(B)$$

However, this is not the case for the following condition:

$$\textit{support}(A \Rightarrow B) \gg \textit{support}(A) \cdot \textit{support}(B) \tag{4.1}$$

This condition can be transformed in an in-equation that is based on the confidence of a rule:

$$\textit{support}(A \Rightarrow B) \gg \textit{support}(A) \cdot \textit{support}(B)$$

$$\frac{\textit{support}(A \Rightarrow B)}{\textit{support}(A)} \gg \textit{support}(B)$$

$$\frac{\textit{frequency}(A \Rightarrow B) \cdot \mid \mathcal{D} \mid}{\textit{frequency}(A) \cdot \mid \mathcal{D} \mid} \gg \textit{support}(B)$$

$$\frac{\textit{support}(A \Rightarrow B)}{\textit{support}(A)} \gg \textit{support}(B)$$

$$\textit{confidence}(A \Rightarrow B) \gg \textit{support}(B) \tag{4.2}$$

However, we can determine an upper bound for *support*(B). It is obvious that the support is highest for a singleton B. Table 4.1 on the next page contains the most frequently changed files

Project (#Txs[4])	File	Count	Max. Support
ECLIPSE	org.eclipse.jdt.core/buildnotes_jdt-core.html	2,403	0.054
(44,786 Txs.)	org.eclipse.jdt.debug/buildnotes_jdt-debug.html	1055	
	org.eclipse.debug.core/buildnotes_platform-debug.html	548	
	org.eclipse.ant.core/buildnotes_platform-ant.html	357	
	org.eclipse.jdt.ui/buildnotes_jdt-ui.html	314	
GCC	gcc/gcc/ChangeLog	21,261	0.463
(45,983 Txs.)	gcc/gcc/version.c	4,361	
	gcc/gcc/cp/ChangeLog	3,913	
	gcc/gcc/testsuite/ChangeLog	3,370	
	gcc/libf2c/libI77/Version.c	3,265	
GIMP	gimp/ChangeLog	5,795	0.660
(8,783 Txs.)	gimp/po/ChangeLog	884	
	gimp/po-plug-ins/ChangeLog	554	
	gimp/configure.in	476	
	gimp/po-script-fu/ChangeLog	285	
JBOSS	build/jboss/build.xml	459	0.040
(11,543 Txs.)	jboss/build.xml	217	
	jboss/src/etc/conf/default/jboss-service.xml	151	
	contrib/jetty/build.xml	142	
	jboss/src/etc/conf/default/standardjbosscmp-jdbc.xml	135	
JEDIT	jEdit/doc/TODO.txt	578	0.304
(1,905 Txs.)	jEdit/doc/CHANGES.txt	560	
	jEdit/org/gjt/sp/jedit/textarea/JEditTextArea.java	208	
	jEdit/org/gjt/sp/jedit/jedit_gui.props	184	
	jEdit/org/gjt/sp/jedit/jEdit.java	143	
KOFFICE	koffice/kword/kwview.cc	990	0.051
(19,781 Txs.)	koffice/kpresenter/kpresenter_view.cc	866	
	koffice/kword/kwtextframeset.cc	685	
	koffice/kspread/kspread_view.cc	592	
	koffice/kword/kwdoc.cc	592	
PYTHON	python/dist/src/Misc/NEWS	1,022	0.036
(28,802 Txs.)	python/dist/src/configure.in	484	
	python/dist/src/Python/ceval.c	372	
	python/dist/src/Objects/typeobject.c	340	
	python/dist/src/Doc/Makefile	309	
POSTGRESQL	pgsql-server/doc/TODO	1,050	0.082
(12,894 Txs.)	pgsql-server/src/backend/parser/gram.y	410	
	pgsql-server/src/bin/pg_dump/pg_dump.c	277	
	pgsql-server/configure	264	
	pgsql-server/src/backend/postmaster/postmaster.c	263	

Table 4.1: Most Frequently Changed Files

and the resulting upper bound for *support*(B). In most cases this bound is well below 10%. Thus, using a confidence threshold of at least 10% ROSE is on the safe side and can set the significance test aside. For the projects with a bound above 10%, ROSE will likely recommend entities that are not statistically significant, but are still justified (e.g., TODO.txt for JEDIT).

4.2 Association Rules in ROSE

4.2.1 Choosing the Task-Relevant Data

ROSE searches for patterns in the task-relevant data $\mathcal{D}$ which consists of past transactions that have been inferred during preprocessing. Patterns evolve during time; one pattern being important may become less important or even incorrect later in a project. In addition, the set of entities constantly evolves, too (functions are added or removed). Thus, the concept of patterns softens.

These problems can be addressed using two different approaches:

Mine from transactions of the last, say, year. Thus, only new or constantly occurring patterns are found. Additionally, new patterns establish fast because strong old patterns are discarded.

Rate new transactions higher than old ones. This approach introduces an implicit aging for rules. Still, all patterns (old and new ones) are found, but usually new or constantly occurring patterns are rated higher than old ones.

Currently, ROSE applies none of these techniques because the main focus of this work is on general predictiveness. The realization of the approaches above will be future work.

4.2.2 From Rules to Recommendations

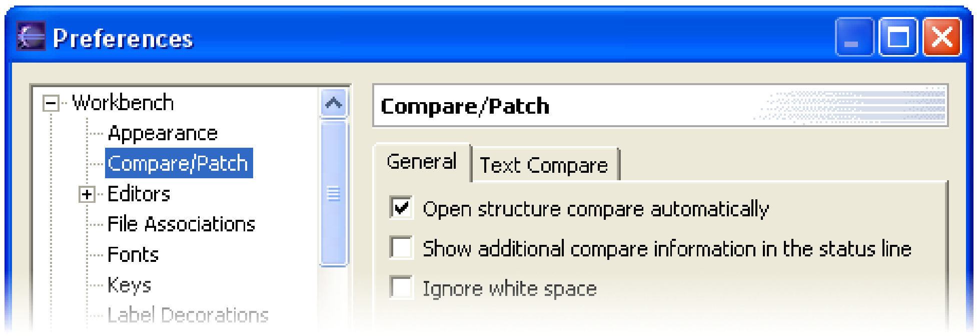
As soon as the programmer begins to make changes, the ROSE client suggests possible further changes. This is done by *applying* matching rules. In general, two notions of matching rules exist:

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

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

For both notions, the antecedent of a rule is satisfied, but only for strong matching it is satisfied exactly. We refer to the set of items Σ as the *situation* in which ROSE makes recommendations. Recall that an item is an action, e.g., change, and an entity.

[4]Table 4.1: Number of transactions is *after* data cleaning.



✓ Open structure compare automatically

☐ Show additional compare information in the status line

☐ Ignore white space

Figure 4.1: An Example for an ECLIPSE preference

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

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

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

Recall Figure 1.2 on page 3; assume the task of a programmer is to extend ECLIPSE [5] with a new preference. Usually, a preference consists of GUI elements, a default value, and a description (see the example in Figure 4.1). In Figure 1.2 the programmer has extended the array fKeys[] in file ComparePreferencePage.java. Thus, the situation Σ_1 is:

$$\Sigma_1 = \{\text{change}(\mathsf{fKeys[]})\}$$

ROSE finds many matching rules for this situation; one of them is r:

$$\text{change(fKeys[])} \Rightarrow \text{change(initDefaults())} \wedge \text{change(plugin.properties)} \qquad [7; 0.875]$$

Using the rule r in the given situation Σ_1, ROSE suggests the consequent of r:

$$\mathit{apply}_{\{r\}}(\Sigma_1) = \{\text{change}(\mathsf{initDefaults()}), \text{change}(\mathsf{plugin.properties})\}$$

The entire set $\mathcal{R}$ of actually mined rules contains further rules, though. The actual result of $\mathit{apply}_{\mathcal{R}}(\Sigma_1)$ is shown in Figure 1.2, ordered by confidence. In practice, ROSE uses all strong rules for recommendations.

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

[5]It is important to capture that ROSE is used in *and* on ECLIPSE.

changes the method initDefaults(). Again, ROSE proposes additional changes which are in this case the same as before, except that now initDefaults() is missing. The situation now additionally contains initDefaults():

$$\Sigma_2 = \{\text{change}(\text{fKeys[]}), \text{change}(\text{initDefaults()})\}$$

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 page where preferences can be set (in Figure 4.1 page *General* is open). Now, she extends the createGeneralPage() method, resulting in a new situation Σ_3:

$$\Sigma_3 = \{\text{change}(\text{fKeys[]}), \text{change}(\text{initDefaults()}), \text{change}(\text{createGeneralPage()})\}$$

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

$$\begin{array}{ll}
\Sigma_3 \Rightarrow \{\text{change}(\text{plugin.properties})\} & [5; 1.0] \\
\Sigma_3 \Rightarrow \{\text{change}(\text{TextMergeViewer()})\} & [3; 0.6] \\
\Sigma_3 \Rightarrow \{\text{change}(\text{propertyChange()})\} & [3; 0.6] \\
\Sigma_3 \Rightarrow \{\text{change}(\text{build.html})\} & [3; 0.6]
\end{array}$$

Applying the above rules yields the union of the consequents of all rules because they have the same antecedent and match the situation Σ_3. ROSE will rank the entities by their confidence, suggesting the user to change the file plugin.properties next. This file contains the descriptions that are used for the labels of a preference (e.g., “Open structure compare automatically” in Figure 4.1).

The next two sections present mining techniques: The *Apriori algorithm* mines for *all strong* rules; the ROSE approach in contrast mines only for *strong* and *matching* rules. Whether rules are matching or not depends on the situation Σ in which ROSE is called.

4.3 The Apriori Approach for Mining Association Rules

One of the most popular approaches for mining *all strong* association rules is the Apriori algorithm [AS94, MTV94]. It takes a *min_supp* and a *min_conf* threshold and the task-relevant data $\mathcal{D}$ as an input[6].

Internally, the Apriori algorithm represents patterns with *itemsets*[7]. A k-itemset is an itemset of size k. An itemset is called *frequent* if it satisfies the support (or frequency) threshold. The set of all frequent k-itemsets is denoted as L_k.

The *Apriori property* helps to reduce the search space for frequent itemsets:

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

[6]The threshold for the frequency *min_freq* is computed using *min_supp* and $\mid \mathcal{D} \mid$.

[7]In data mining literature item sets are spelled as itemsets.

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

The Apriori algorithm consists of two phases:

1. **Find all frequent itemsets.**
   Frequent itemsets are generated level-wise: First L_1 is computed, then L_1 is used to find L_2 which is used to compute L_3, and so on. This phase terminates if for a k no more frequent k-itemsets are found. Each level, i.e., the creation of a set L_k, consists of four steps:

   – The *join* step:
   A candidate k-itemset C_k is generated by joining L_{k-1} with itself. The join condition is that the first $k - 1$ items of two itemsets l_1 and l_2 are equal and only the last elements differ: $l_1[k] < l_2[k]$.

   – The *prune* step:
   Remove itemsets from C_k that cannot be frequent by means of the Apriority property. The check whether subsets are frequent or not can be done quickly by maintaining a hash tree of all frequent itemsets.

   – The *scan* or *count* step:
   Scan the database $\mathcal{D}$ and count the frequency of each remaining candidate in C_k.

   – The *create* step:
   The frequent k-itemsets L_k are those sets in C_k that satisfy the frequency threshold.

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

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

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

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

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

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

Figure 4.2 on the following page shows an example for the Apriori algorithm. The candidate 1-itemset C_1 corresponds to the set of all items $\mathcal{I}$. The count step reveals that itemset $\{E\}$ is not frequent. Next the candidate 2-itemsets C_2 are generated by joining L_1 with itself (C_2 is

Relevant transactions:

$\mathcal{D}$

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

Generate frequent 1-itemset L_1

$\longrightarrow$

C_1

Itemset
$\{A\}$
$\{B\}$
$\{C\}$
$\{D\}$
$\{E\}$

$\xrightarrow{\textbf{count}}$

C_1

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

$\xrightarrow{\textbf{create}}$

L_1

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

Generate frequent 2-itemset L_2

$\xrightarrow{\textbf{join}}$

C_2

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

$\xrightarrow{\textbf{prune}}$

C_2

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

$\xrightarrow{\textbf{count}}$

C_2

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

$\xrightarrow{\textbf{create}}$

L_2

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

Generate frequent 3-itemset L_3

$\xrightarrow{\textbf{join}}$

C_3

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

$\xrightarrow{\textbf{prune}}$

C_3

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

$\xrightarrow{\textbf{count}}$

C_3

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

$\xrightarrow{\textbf{create}}$

L_3

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

Generate association rules from L_2 and L_3

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

Figure 4.2: An Example for the Apriori Algorithm (*min_freq* $= 2$; *min_conf* $= 0.5$)

always the cross product of L_1). For $k = 2$ it is never possible to prune any elements because all subsets are singletons and always contained in L_1. The count step identifies $\{B, D\}$ and $\{C, D\}$ as not frequent. Next the candidate 3-itemsets C_3 are generated from L_2 using the join condition $l_1[1] = l_2[1] \wedge l_1[2] > l_2[2]$. This returns three itemsets. Two of them are not frequent by the Apriori property and pruned: For $\{A, B, D\}$ the subset $\{B, D\}$ is not frequent and for the itemset $\{A, C, D\}$ subset $\{C, D\}$ is not frequent. For the third candidate $\{A, B, C\}$ a database scan verified that it is frequent. After all frequent itemsets have been computed, each itemset in L_2 and L_3 is used to create rules. The confidence is computed for each rule and only strong rules are returned.

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

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

It is possible to mine association rules without candidate generation based on a divide-and-conquer strategy. The algorithm is called *frequent-pattern growth* and also known as *FP-growth* [HPY00].

4.4 The ROSE Approach for Mining Association Rules

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

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

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

Of course, too low support thresholds have a bad influence on the quality of recommendations. Nevertheless, the developer should be able to decide on support thresholds and not any technical boundaries created by the Apriori algorithm.

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

Therefore, ROSE uses its own mining algorithm that mines *only required* rules *on the fly*. This algorithm is based on two optimizations:[8]

Mine with constrained antecedents. In our specific case, the antecedent is equal to the situation; hence, we only mine rules *on the fly* which *match* the situation Σ, i.e., rules that are $\Sigma \Rightarrow B$. Mining with such constrained antecedents [SVA97] takes only a few seconds. An additional advantage of this approach is that it is incremental in the sense that it allows new transactions to be added to $\mathcal{D}$ between two situations. Thus, recommendations are always up-to-date.

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

The ROSE mining algorithm consists of three steps:

Find transactions. Find all transactions T that contain *all* items of the situation Σ, i.e., $\Sigma \subseteq T$. Using the database schema of Chapter 3 and relation algebra, we denote these transactions as $\sigma_{\mathit{TransactionID}}(\mathit{Lineitems} \div \Sigma)$.

Group & sort. Group the items $\mathit{Lineitems} \bowtie \sigma_{\mathit{TransactionID}}(\mathit{Lineitems} \div \Sigma)$ of these transactions by *EntityID*, and sort them by their descending count.

Create rules. Each group corresponds to exactly one single-consequent rule.

- The frequency of Σ is the maximal count of a group (which is likely for an group of an item $i \in \Sigma$ and is always for the first returned group).
- The count for a group of an item i is the frequency for the rule $\Sigma \Rightarrow \{i\}$.
- The confidence of a rule $\Sigma \Rightarrow \{i\}$ is

$$\frac{\mathit{frequency}(\Sigma \Rightarrow \{i\})}{\mathit{frequency}(\Sigma)}$$

- Ignore *trivial* rules—that are rules $\Sigma \Rightarrow \{i\}$ with $i \in \Sigma$.

Return only rules that satisfy the support and confidence thresholds.

Figure 4.3 on the next page shows an example for the ROSE mining algorithm. Suppose, the situation is $\Sigma = \{A, B\}$. First, ROSE searches all transactions that contain Σ: 100, 300, and 700. Next, it groups exactly those transactions by items and sorts them by their descending count. The highest count is for item A, thus the *frequency* for Σ is 3. The rules for A and B are trivial

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

Generate frequent k-itemsets and $k+1$-itemsets that contain Σ

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

$\xrightarrow{\textbf{find}}$

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

$\xrightarrow{\textbf{group \& sort}}$

Item	Frequency	
A	3	⇒ {A, B }
B	3	⇒ {A, B }
C	2	⇒ {A, B, C }

Create single-consequent rules with antecedent Σ

Item	Frequency	
A	*frequency*(Σ) =3	{A,B} ⇒ {A} is trivial
B	3	{A,B} ⇒ {B} is trivial
C	2	{A,B} ⇒ {C} has *frequency* =2, *confidence* =2/3 and is strong

Figure 4.3: An Example for the ROSE Algorithm (*min_freq* $= 2$; *min_conf* $= 0.5$)

(because both are in the situation), thus they are ignored. For C, the rule $\Sigma = \{A, B\} \Rightarrow \{C\}$ is strong because the thresholds for *min_freq* and *min_conf* are satisfied.

The optimizations above make mining very efficient: The average runtime of a query is about 0.5s for large version histories like GCC.[9]

ROSE provides another mining algorithm for *single antecedent single consequent* rules $\{a\} \Rightarrow \{b\}$. Such rules are less precise for recommendations, but valuable for measurement and visualization of coupling between entities (see Section 5.1 for some examples). The algorithm is exactly like the Apriori algorithm presented in Section 4.3, except that only 2-frequent itemsets are generated and used for rule creation.

In the next chapter we will present some examples for association rules.

[8]These optimizations have already been used in the example of Subsection 4.2.2.

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

Chapter 5

Real Life Examples

The young men know the rules.
The old men know the exceptions
– Oliver Wendell Holmes

This chapter gives anecdotic evidence for the usefulness of ROSE. We start with the most basic kind of rules called *single antecedent single consequent* rules. Then we show the superiority of *single consequent rules*. We will see in the last section that ROSE is not restricted to specific programming languages—it even detects relations between source code and text files.

5.1 Visualization of Binary Coupling

Single antecedent single consequent rules (or sometimes referred to as *binary* rules) are the most simple kind of association rules. They are implications of the form $\{a\} \Rightarrow \{b\}$ with $a, b \in \mathcal{I}$.

Their value for recommendations is limited because they cannot take advantage of the situation Σ. This means whenever a user changes more entities, e.g., e and f, an entity g may be proposed, although the association rule $e \wedge f \Rightarrow g$ has no or to few support. Thus, using such rules, one will get more but less precise recommendations.

However, binary rules are very important for measuring *coupling* between modules, directories, files, or functions (which we subsume as entities): Two elements are coupled if they occur together in at least one transaction. The stronger the support and confidence values are the stronger is the coupling. Such coupling by simultaneous changes is referred to as *logical coupling* [GJKT97, GJK03]. However, we will prefer the term *evolutionary coupling* because it clearly states the domain of this kind of coupling [ZDZ03].

Evolutionary coupling can be represented as a graph $G = (V, E)$, with the entities as nodes $V = \mathcal{E}$ and for each coupling between to entities e and f an edge $(e, f) \in E$. We will call these graphs *coupling graphs*. However, drawing such huge graphs will be expensive and provides only unsatisfying results.

One special layout for coupling graphs are *pixel-maps*. The idea is to visualize not the graph

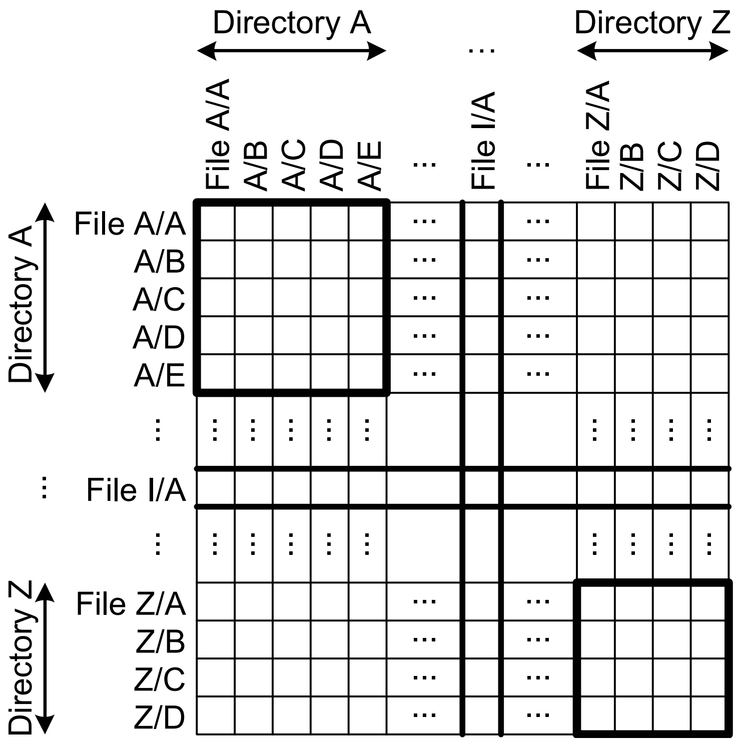


Figure 5.1: An Example for a Pixel-Map

itself, but its adjacency matrix. Each edge is thus represented by a single *pixel* and each node as two lines (one horizontal and one vertical). Thus, pixel-maps can easily visualize large graphs. Figure 5.1 shows an abstract example for an pixel-map. For instance, file A/C is represented by the third row and the third column.

Another advantage of pixel-maps is that using a specific order, e.g., lexicographically by directory and file name, we can emphasize a given structure. For instance the boldfaced blocks in Figure 5.1 represent all coupling within directories. Everything outside these blocks is coupling between different directories.

An important aspect is the use of color in pixel-maps. We can color a pixel (x, y) by:

- the *support* or *frequency* of the rule $x \Rightarrow y$,
- the *confidence* of the rule $x \Rightarrow y$, or
- the *correlation* of the rule $x \Rightarrow y$ which is:

$$\mathit{correlation}(x \Rightarrow y) = \frac{\mathit{support}(x \Rightarrow y)}{\mathit{support}(x) \cdot \mathit{support}(y)}$$

Note that support pixel-maps are symmetric. Confidence and correlation pixel-maps are not symmetric: The pixel (x, y) usually has a different color than (y, x). All pixels (x, x) on the top-down diagonal of a confidence pixel-map have the same confidence value $\mathit{confidence}(x \Rightarrow x) = 1.0$ and thus the same color.

In the remainder we will focus on confidence pixel-maps. The color ranges from *blue*, for low confidence values, to *red*, for high confidence values.

5.1.1 Coupling within DDD

Figure 5.2 on the next page shows such a confidence pixel-map for the DDD debugger. Each pixel (x, y) represents the coupling between two files x and y. The files have been sorted lexicographically by directory and file name.

How do we read such pixel-maps? Basically, we look for *presence* and *absence* of coupling. This coupling may have different forms:

Pixels—Coupling between files. This is the most basic kind of coupling.

Lines—Coupling between a files and a directory. In Figure 5.2 the ddd/-directory consists of two parts: the *source code* and *pictures* (in directory PICS). One can easily spot four lines (for this example we take symmetry into account and consider only lines below the diagonal):

- the *upper horizontal line* is the file DDD.mk.in
- the *lower horizontal line* is the file Makefile.bin
- the *left vertical line* is the file PICS/ddd-graph.eps
- the *right vertical line* is the file PICS/FIX-XPM

Thus DDD.mk.in and Makefile.bin are coupled with the PICS/-directory; and the source code is coupled with PICS/ddd-graph.eps and PICS/FIX-XPM.

Blocks—Coupling between directories. Figure 5.2 contains several blocks aligned to the diagonal, e.g., the source code block or pictures block in directory ddd/. These blocks represent coupling within one directory.

But Figure 5.2 also shows coupling between different directories, e.g., the block labeled *Tests* represents coupling between the *test*-directory and the source code.

Concentrating on the absence of blocks, we will notice that most external *libraries* are not coupled by evolution with the other parts of DDD.

5.1.2 Coupling within ECLIPSE

Figure 5.3 on the following page shows a pixel-map for the JAVA debugging component of ECLIPSE. We can spot four independent parts that match different plug-ins of ECLIPSE:

- the *org.eclipse.jdt.debug.jdi.tests* plug-in:
  Interestingly, almost no coupling exists within this plug-in.
- the *org.eclipse.jdt.debug.tests* plug-in.

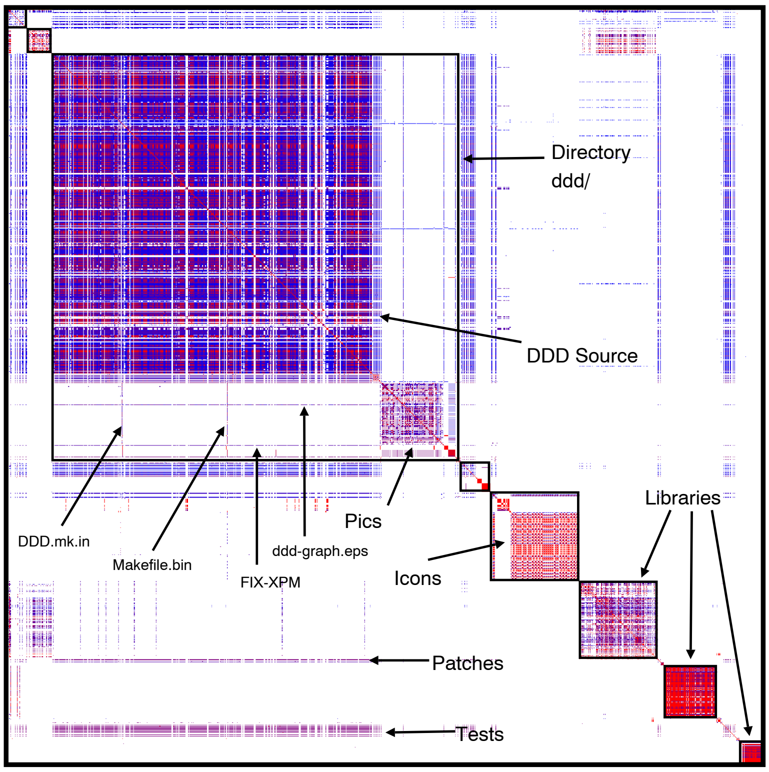


Figure 5.2: A Confidence Pixel-Map for DDD

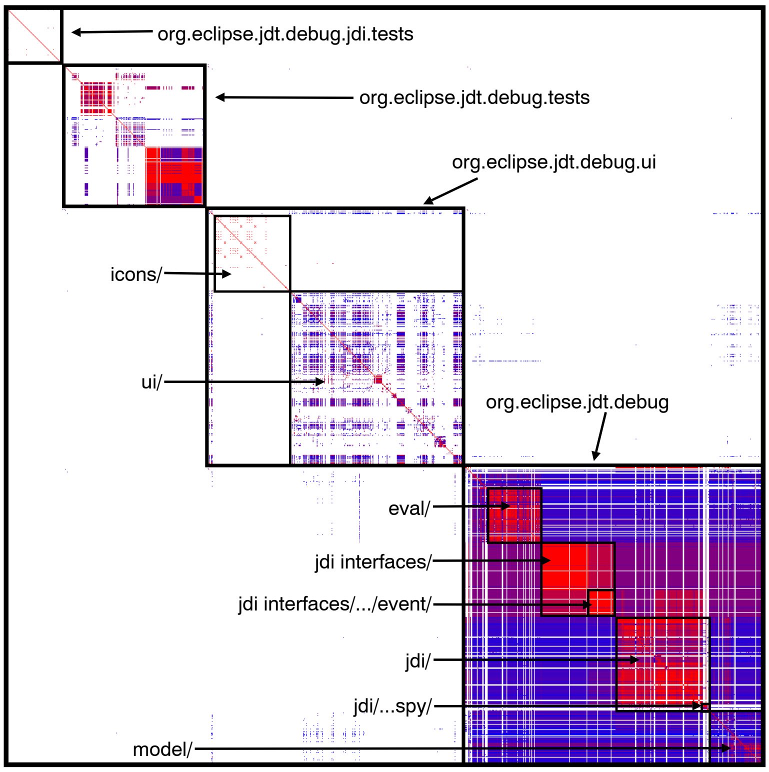


Figure 5.3: A Confidence Pixel-Map for the JAVA Debugging Component of ECLIPSE

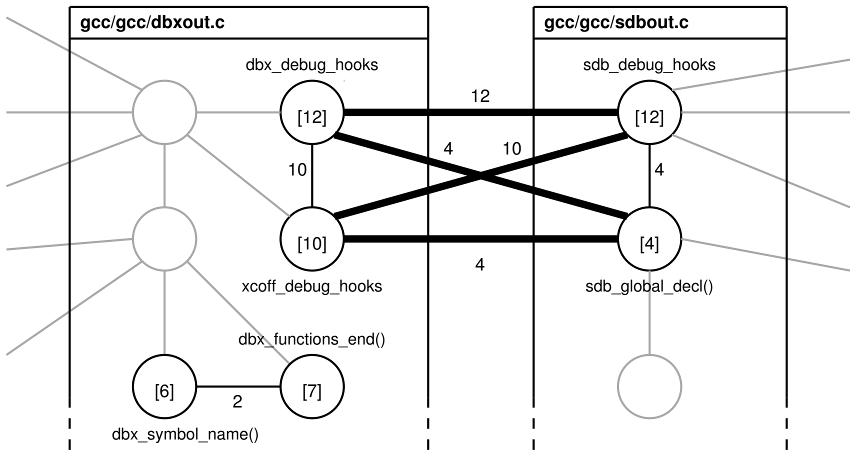


Figure 5.4: Evolutionary Coupling between Debugging Symbols in GCC

- the *org.eclipse.jdt.debug.ui* plug-in:
  This plug-in consists of two independent parts: the *icons* and the actual *user interface*.
- the *org.eclipse.jdt.debug* plug-in:
  Almost everything within this plug-in is coupled. But the strong (red) coupling appears near the diagonal in subdirectories called eval/, jdi interfaces/, and jdi/. Usually this is an indicator for good architecture.

  Note that there exists some coupling between the model/ directory and the ui/ directory of the *org.eclipse.jdt.debug.ui* plug-in.

5.2 Association Rules Increase Clarity

5.2.1 Debugging Symbols in GCC

Consider Figure 5.4, visualizing the coupling graph of some program entities in the GNU Compiler Collection (GCC). We see two files dbxout.c and sdbout.c (square rectangles) that issue debugging symbols in DBX and SDB format, respectively.

Both files contain some entities, depicted as vertices—*variables* such as dbx_debug_hooks in dbxout.c and *methods* such as sdb_global_decl(). The numbers in brackets show how frequently the entity has been changed over the revision history of GCC—xcoff_debug_hooks, for instance, has been changed ten times. The number associated with each edge indicates *how often* the related entities have been changed together. So, we can see that

- In all 12 cases where dbx_debug_hooks was changed, so was sdb_debug_hooks, and vice versa.
- In all 4 cases where sdb_global_decl() was changed, so were the other debug_hooks variables—in both files.

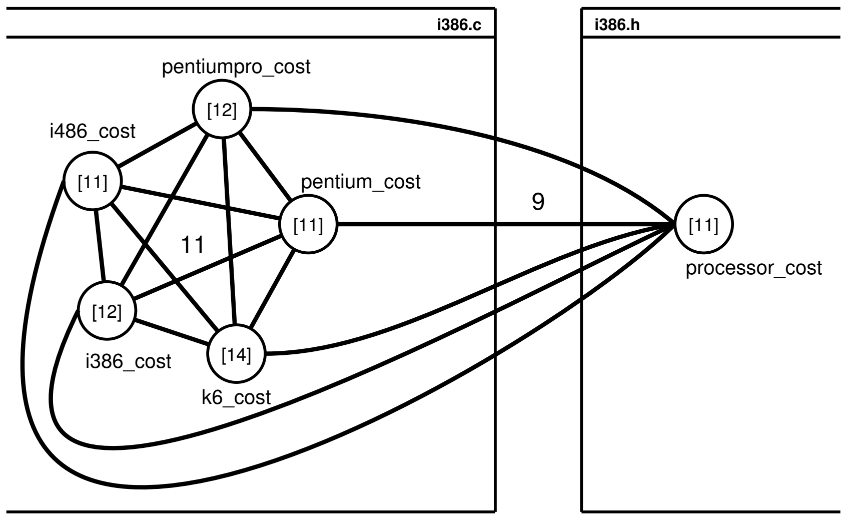


Figure 5.5: Evolutionary Coupling between Processor Costs in GCC

- dbx_functions_end() and dbx_symbol_name() have been changed together, but never with an entity in sdbout.c.

However, such graphs are difficult to read, expecially if one is interested in multiple files. For instance, we can express the relation between the files dbxout.c and sdbout.c using two unlimited association rules instead of four binary rules:

change(dbx_debug_hooks) $\wedge$ change(xcoff_debug_hooks)
$\Rightarrow$ change(sdb_debug_hooks) $\wedge$ change(sdb_global_decl)
[*frequency*=4, *confidence*=0.40]

change(sdb_debug_hooks) $\wedge$ change(sdb_global_decl)
$\Rightarrow$ change(dbx_debug_hooks) $\wedge$ change(xcoff_debug_hooks)
[*frequency*=4, *confidence*=1.00]

5.2.2 Processor Costs in GCC

GCC has arrays that define the costs of different assembler operations for INTEL processors: i386_cost, i486_cost, pentium_cost, pentiumpro_cost, and k6_cost. These have been changed together in 11 transactions. In 9 of these 11 transactions, this change was triggered by a change in the type processor_costs. Figure 5.5 shows the corresponding binary relations.

With association rules the relation between the cost arrays and their type will be expressed more clearly. For this example, three possible frequent itemsets are:

$$\begin{aligned} F_1 &= \{\text{change}(\text{processor\_cost})\} \\ F_2 &= \{\text{change}(\text{i386\_cost}), \text{change}(\text{i486\_cost}), \text{change}(\text{k6\_cost}), \\ &\quad \text{change}(\text{pentium\_cost}), \text{change}(\text{pentiumpro\_cost})\} \\ F_3 &= F_1 \cup F_2 \end{aligned}$$

We can create the following association rule using the itemsets F_1 and F_3:

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

So, whenever the costs type is changed (e.g., extended for a new operation), ROSE suggests to extend the appropriate cost instances, too.[1]

5.3 ROSE Mines Everything

ROSE also detects coupling that is out of reach for most program analyes:

Coupling between different programming languages.

ROSE is not restricted to a specific programming language. In fact, it can detect coupling between program parts written in different languages. Here is an example, taken from the PYTHON library:

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

Whenever the C file _Qdmodule.c was changed, so was the PYTHON file qdsupport.py—a classical coupling between interface and implementation—ROSE even detects the affected functions GrafObj_getattr() and outputGetattrHook(). Detecting such coupling might be possible with program analysis, but will be very complex.

Coupling between source code and non-source code.

Recall the preference example from the Chapter 4:

$$\begin{aligned}\text{change(fKeys[])} \Rightarrow \text{change(initDefaults())} \wedge \text{change(plugin.properties)} \\ [\textit{frequency}=7; \textit{confidence}=0.875]\end{aligned}$$

Whenever a programmer extends ECLIPSE with a new preference (in fKeys[]), she also has to set a default value (in initDefaults() and a description for the user interface in plugin.properties.

Coupling between source code and text files (like plugin.properties) is undetectable by program analysis. More than 12,000 of the 27,000 ECLIPSE files are configuration, build, documentation, and images files and thus out of reach for program analysis.

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

Coupling between documentation.

ROSE can also reveal coupling between items that are not even programs, as in the POSTGRESQL documentation:

$$\begin{aligned}&\text{change}(\mathsf{createuser.sgml}) \land \text{change}(\mathsf{dropuser.sgml})\\ &\Rightarrow \text{change}(\mathsf{createdb.sgml}) \land \text{change}(\mathsf{dropdb.sgml})\\ &\qquad\qquad [\textit{frequency} = 11; \textit{confidence} = 1.00]\end{aligned}$$

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

The next chapter will give empirical evidence for the usefulness of ROSE.

Chapter 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. Can ROSE find the missing one?

Closure. Suppose a transaction is finished and 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 Setting

For our evaluation, we analyzed the archives of eight large open-source projects (Table 6.1 on the following 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 T whether its items can be *predicted from earlier history:*

1. We create a *test case* $q = (Q, E)$ consisting of a *query* $Q \subseteq T$ and an *expected outcome* $E = T - Q$.
2. We take all transactions T_i that have been completed before *time*(T) as a *training set* and mine a set of rules R from these transactions.
3. To avoid having the user work through endless lists of suggestions, ROSE only shows the *top ten single-consequent rules* $R_{10} \subseteq R$ ranked by confidence. In our evaluation, we

[1]Table 6.1: Number of transactions is *before* data cleaning.

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

Table 6.1: Analyzed Projects (Txn = Transaction; Ety = Entity)

apply R_{10} to get the result of the query $A_q = \textit{apply}_{R_{10}}(Q)$. Thus, the size of A_q is always less or equal than ten.

4. The result A_q of a test case q consists of two parts:

   - $A_q \cap E_q$ are the items that *matched* the expected outcome and are considered *correct* predictions.
   - $A_q - E_q$ are unexpected recommendations that are considered as *wrong* predictions and called *false positives*.

   Additionally, ROSE may have missed items:

   - $E_q - A_q$ are *missing* predictions and called *false negatives*.

   The sets A_q and E_q are illustrated in Figure 6.1 on the next page.

For the assessment of a result A_q, we use two measures from information retrieval [Rij79]: The *precision* P_q describes which fraction of the returned items was actually correct, i.e., expected by the user—the higher the precision the fewer the false positives. The *recall* R_q indicates the percentage of correct predictions—the higher the recall the fewer false negatives.

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

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

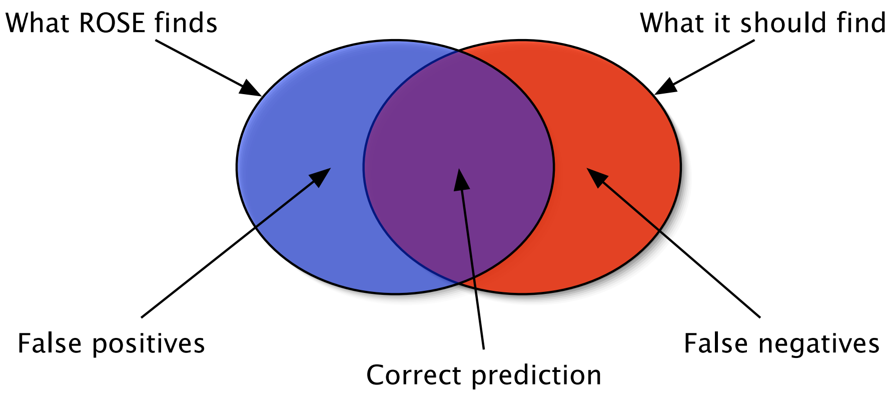


Figure 6.1: Precision and Recall

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

In practice, though, recall and precision correlate negatively with each other. For a high recall, one could return many or even all items resulting in a low precision. On the other hand only recommending a few certain items results in a high precision, but a low recall.

For each query q_i we get a precision-recall pair (P_{q_i}, R_{q_i}). 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:

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

This approach uses the precision and recall values that 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 *per* query.

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

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

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 that *summarize 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. 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.

The difference between macro-evaluation and micro-evaluation is important. Micro-evaluation computes average precision and recall *per item* and *not per query*. This is illustrated in the following example:[2]

> Suppose we have two lectures:
>
> - Lecture A with 100 students of which 25 wear glasses. The ratio for this lecture is thus 25%.
> - Lecture B with 20 students of which 15 wear glasses; resulting in a ratio of 75%.
>
> Averaging these figures with macro- and micro-evaluation we get:
>
> - *Macro-evaluation* takes the average of both ratios:
>
> $$\frac{25\% + 75\%}{2} = 50\%$$
>
>   The average value is calculated on *lecture*-level. This means that a lecture has on average 50% students that were glasses.
> - *Micro-evaluation*, in contrast, calculates the average-value on *student*-level:
>
> $$\frac{100 \cdot 25\% + 20 \cdot 75\%}{100 + 20} = \frac{25 + 15}{120} = 33.3\%$$
>
>   If the students of A and B are disjoint, this means that every third student of these two lectures wear glasses.
>
> This example shows that one has to use and interpret average values very carefully.

As macro-evaluation is misleading in some cases, all averages are given by micro-evaluation, unless otherwise noted. We will also determine the likelihood that the topmost three recommendations contain at least one correct prediction (see Section 6.3 for details).

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

Figure 6.2 on the facing page 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 that have the same support are connected with lines. As a result we get three *precision-recall*

[2]inspired by [SM83]

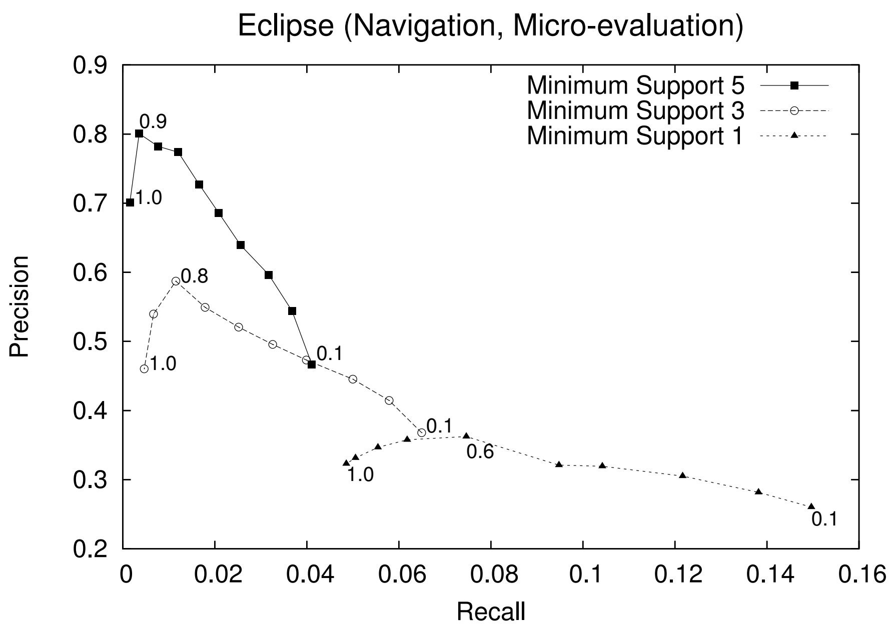


Figure 6.2: Varying Support and Confidence

curves, one for each investigated support. (The connecting lines between measured values are for the sake of clarity and not for interpolation.)

In Figure 6.2, 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). On average, the programmer has to check about four suggestions in order to find a correct one.

Figure 6.2 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 6.2 shows that high support *and* confidence thresholds are required for a high precision. Still, such values result in a very low recall and thus indicate a trade-off between precision and recall.

In practice, a graph such as the one in Figure 6.2 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

A precision like 26% sounds low, but 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, resulting in an average precision of only 33% per recommendation.

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:

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

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

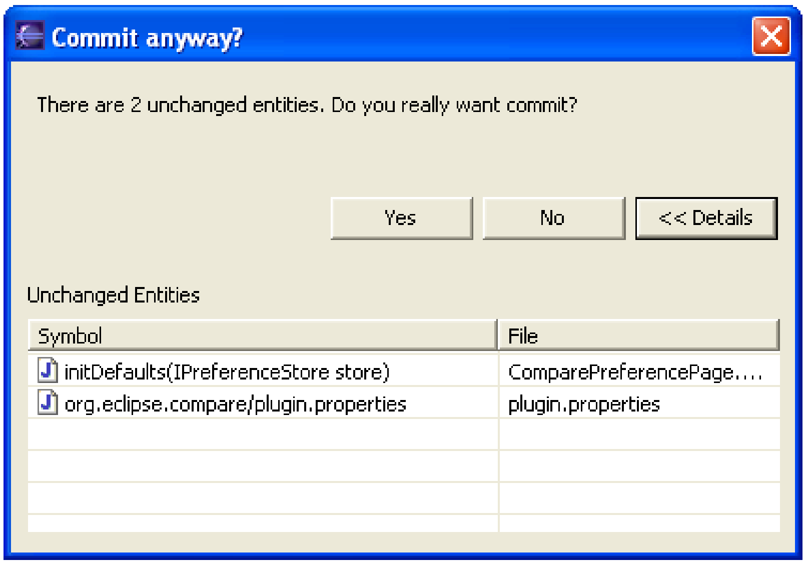
We repeated the experiment from Section 6.2 for all eight projects with a support threshold of 1 and a confidence threshold of 0.1—such that for navigation, the user gets several recommendations. The results are shown in Table 6.2 on page 66 (column *Navigation*). For these settings the average recall is 15%, and 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—in Figure 6.3 on the facing page the developer has missed to changed initDefaults() and plugin.properties.



Commit anyway?

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

Unchanged Entities

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

Figure 6.3: Error Prevention in ROSE

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

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 [3]. Still, we wanted to get as many missing entities as possible, which resulted in a support threshold of 3.

The results are shown in Table 6.2 (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 recognized *and* 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, it predicts in every second case the missing entity.

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 the case that no entity is missing. We simulated this by testing *complete transactions.*

	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 6.2: Results for Fine Granularity (R = Recall; P = Precision; L = Likelihood)

For each transaction T, we queried $Q = T$, and checked whether ROSE would predict $E = \emptyset$. Thus, we 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 6.2 (column *Closure*). We see that 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[4]. 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 exclusively 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 6.3 on the next page. It turns out that the coarser granularity increases recall in *all*

[4]By default, ROSE recommends only complete files if it cannot look inside a file, like for `plugin.properties`. For source code the preferred granularity is on *functions* and *fields,* and for $\TeX$ files on *subsections*.

	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 6.3: Results for Coarse Granularity (R = Recall; P = Precision; L = Likelihood)

cases (sometimes even dramatically, as the factors 3–8 in KOFFICE show). The precision stays comparable or is even increased.

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.[5]

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* [SA95] (also known as *multilevel association rules* [HF95]) 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.

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

[5]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.

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 as well as bad practices. However, we believe that competent programmers make more “good” 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.

Chapter 7

Related Work

7.1 Data Preprocessing

Data extraction from CVS is very well covered, and many tools are available for free: Daniel German and Audris Mockus created SOFTCHANGE [Sof04]—a tool that extracts and summarizes information from CVS and bug tracking databases [GM03]. Dirk Draheim and Lukasz Pekacki developed BLOOF [Blo04] which extracts CVS log data into a database, and visualizes the software evolution using metrics [DP03].

Michael Fischer et al. demonstrated how to populate a *release history database* linking data from CVS and BUGZILLA [FPG03b]. In [FPG03a] they also combined their approach with features. Another project that considers multiple data sources is HIPIKAT by Davor Čubranić and Gail Murphy [ČM03]. They integrate information from CVS, BUGZILLA, and developer mailing lists using text similarity.

To our knowledge, transaction recovery has been used by many approaches but has nowhere been covered in detail: Harald Gall, Daniel German, and Audris Mockus used fixed time windows in the past [GJK03, GM03, MFH02], and we used sliding time windows in our previous work [ZDZ03, ZWDZ04]. Commit mails have not been used in recent work to restore transactions.

Up to now, only a few approaches have considered fine-grained changes: Harald Gall et al. [GJK03] and James Bieman et al. [BAY03] both analyzed relations between classes. In our previous work we applied the approach presented in Section 3.4 and mined for relations [ZDZ03] and association rules [ZWDZ04] between functions, sections, and other fine-grained building blocks.

Michael Fischer et al. also proposed an algorithm for detecting merges of revisions in their release history database paper [FPG03b]. Lijie Zou and Michael Godfrey showed how to use origin analysis to detect merging and splitting of functions in [ZG03]. Nonetheless, data cleaning is often neglected, and there is still much room for improvement.

7.2 Mining in Software Engineering

Independently from us, Annie Ying developed an approach that also uses association rule mining on CVS version archives [Yin03, YMNCC04]. She especially evaluated the usefulness of the results, considering a recommendation most valuable or “surprising” if it could not be determined by program analysis. She 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. A similar work has been done by Ahmed Hassan [iSS04].

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 [MV00]. Various researchers computed metrics on the module or file level [BKPS97, GJKT97, GKMS00, HH03] or orthogonal to these per feature [MWZ03], and investigated the change of these metrics over time, i.e., for different releases or versions of a system.

Harald Gall et al. were the first to use release data to detect logical coupling between modules [GHJ98]. The CVS history allows to detect more fine-grained logical coupling between classes [GJK03], files, and functions [ZDZ03]. None of these works on logical coupling did address its predictive power. Jelber Sayyad-Shirabad et al. use inductive learning to learn different concepts of relevance between logically coupled files [SSLM01, SSLM03, SSLM04]. A concept is a relevance relation, for example whether two files have been update simultaneously. Instances of concepts are described in terms of *attributes* such as file name, extension and simple metrics like number of routines defined. Jelber Sayyad-Shirabad thoroughly evaluated the predictive power of the concepts found, but none of the papers 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 [Mic99] or generalized association rules [Mic00]. 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, Amir Michail mines a single version, while ROSE uses the changes between different versions.

In order to guide programmers, a number of tools have exploited *textual similarity* of log messages [CCW$^+$01] or program code [Atk98]. HIPIKAT [ČM03] by Davor Čubranić improves that by taking also other sources like mail archives and online documentation into account. In contrast to ROSE, all these tools focus on high recall rather than on high precision, and on relationships between files or classes rather than between fine-grained entities.

7.3 Workshop on Mining Software Repositories (MSR)

Mining software repositories is an emerging research area. Therefore, it has been the topic of the MSR workshop that has been co-located with ICSE 2004.

Filip Van Rysselberghe et al. mined for *frequently applied changes* using clone detection technqiues [RD04], and Mohammad El-Ramly et al. mined software usage data, like system-user interaction data [ERS04]. In the area of *defect analysis*, Chadd Williams et al. discussed how to used bug data to find new bugs [WH04], and Thomas Ostrand et al. tried to predict fault-prone files [OW04]. In order to analysis *project communities*, Luis Lopez-Fernandez et al. applied social network analysis to CVS archives [LFRGB04], and Kevin Schneider et al. focused on local interaction data [SGPP04]. Mining for *reuse* seems to be a hot topic in future: Frank McCarey et al. applied collaborative filtering to recommend reuse [MCK04], and Yuhanis Yusof et al. mined for code templates [YR04].

Most of this work is out of scope for ROSE at the moment. However, it will be important for future improvements on ROSE.

Chapter 8

Conclusion

Imagination is more important than knowledge.
Knowledge is limited. Imagination encircles the world.
– Albert Einstein

ROSE can be a helpful tool to suggest further changes to be made and to warn about missing changes. However, 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:

Aspect identification. If program entities have been changed together several times, the common abstractions behind the individual changes may be candidates for *aspects* (as in aspect-oriented programming). An evolutionary coupling would then be turned into a single syntactic entity, such that future changes can be made in one place only.

Rule presentation. The rules detected by ROSE describe the evolutionary software process—which may or may not be the intended process. Consequently, these rules can and should be made explicit. In previous work [ZDZ03], we used visual mining to detect regularities and irregularities of logically coupled items. Such visualizations could further explain the recommendations to programmers and managers.

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”). They may be less precise in the location, but provide higher confidence.

Sequence rules. Right now, we only relate 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. Knowledge about the semantics of programs could also help to separate related changes into likely and non-likely. Furthermore, coupling that can be found via analysis [Yin03] need not be repeated in ROSE’s suggestions.

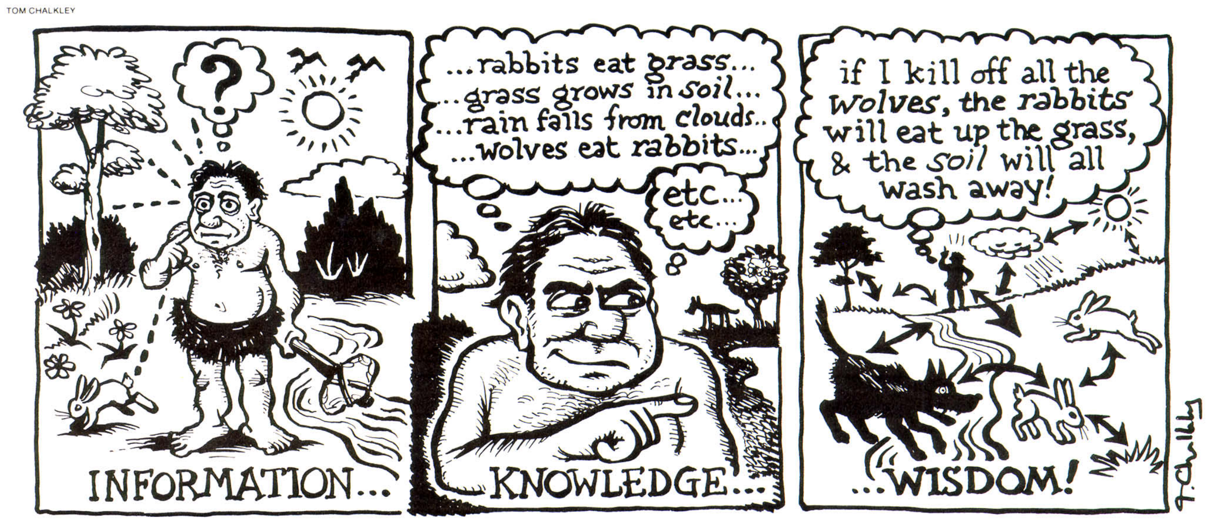
From locations to actions. By combining version histories and program analysis it will be possible to learn patterns, like whenever a developer uses a logger she first imports the logger class and declares an instance.

In other words, after the user types `log.info("Hello World")` we recommend to insert `import logger.Logger` and `Logger log = Logger.createLogger(#className)`. Such recommendations can be integrated in IDEs and executed at the touch of one button.

Currently, some simple code assist features based on program analysis exist. Leveraging version archives we can automatically learn more complex features and additionally verify existing ones. The learning can take place at the user’s place thus enabling developer or project specific patterns.

However, ROSE will move closer to *wisdom*, but never actually will be wise. Thus it will relieve programmer’s work, but not make them unemployed. (For the difference between information, knowledge, and wisdom read the cartoon in Figure 8.1).

At the dawn of the last century, the philosopher George Santayana famously remarked that those who do not learn from history would be condemned to repeat it. Those who do learn from history, though, get the *chance* to repeat it—and this is what our approach provides.


TOM CHALKLEY
?
...rabbits eat grass... ..grass grows in soil... ...rain falls from clouds.. ...wolves eat rabbits...
etc... etc...
if I kill off all the wolves, the rabbits will eat up the grass, & the soil will all wash away!
INFORMATION...
KNOWLEDGE...
...WISDOM!
T. Chalkley

The information-knowledge-wisdom hierarchy. The caveman (left) has lots of information (facts and ideas); he selects and organizes useful information into knowledge (center), but he does not achieve wisdom until he has integrated his knowledge into a whole that is more useful than the sum of its parts.

Figure 8.1: The Information-Knowledge-Wisdom Hierarchy (taken from [Cle82])

Bibliography

[All02] Joshua Allen. Clean underpants with that book? Better Living Through Software Weblog, http://www.netcrucible.com/blog/2002/11/09.html#a250, November 2002.

[AS94] 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.

[Atk98] David 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.

[BAY03] James M. Bieman, Anneliese A. Andrews, and Helen 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.

[BKPS97] Thomas Ball, Jung-Min Kim, Adam A. Porter, and Harvey P. Siy. If your version control system could talk.... In *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*, 1997.

[Blo04] Bloof. Project homepage. http://bloof.sourceforge.net/, June 2004.

[CCW$^+$01] Annie Chen, Eric Chou, Joshua Wong, Andrew Y. Yao, Qing Zhang, Shao Zhang, and Amir Michail. CVSSearch: searching through source code using CVS comments. In *Proc. International Conference on Software Maintenance (ICSM 2001)*, pages 364–374, Florence, Italy, November 2001. IEEE.

[Cle82] Harlan Cleveland. Information as a resource. *The Futurist*, pages 34–39, December 1982.

[ČM03] Davor Čubranić and Gail C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proc. 25th International Conference on Software Engineering (ICSE)*, pages 408–418, Portland, Oregon, May 2003.

[CVS04] CVS. Concurrent versions system, project homepage. https://www.cvshome.org/, June 2004.

[DP03] Dirk Draheim and Lukasz Pekacki. Process-centric analytical processing of version control data. In *Proc. International Workshop on Principles of Software Evolution (IWPSE 2003)*, Helsinki, Finland, September 2003. IEEE Press.

[ERS04] Mohammad El-Ramly and Eleni Stroulia. Mining software usage data. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, pages 64–68, Edinburgh, Scotland, UK, May 2004.

[FO02] Karl Fogel and Melissa O'Neill. *cvs2cl.pl: CVS-log-message-to-ChangeLog conversion script*, September 2002. http://www.red-bean.com/cvs2cl/.

[FPG03a] Michael Fischer, Martin Pinzger, and Harald Gall. Analyzing and relating bug report data for feature tracking. In *Proc. 10th Working Conference on Reverse Engineering (WCRE 2003)*, Victoria, British Columbia, Canada, November 2003. IEEE.

[FPG03b] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proc. International Conference on Software Maintenance (ICSM 2003)*, Amsterdam, Netherlands, September 2003. IEEE.

[GHJ98] Harald Gall, Karin Hajek, and Mehdi 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, November 1998. IEEE.

[GJK03] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. CVS release history data for detecting logical couplings. In *Proc. International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 13–23, Helsinki, Finland, September 2003. IEEE Press.

[GJKT97] H. Gall, M. Jazayeri, R.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.

[GKMS00] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7), 2000.

[GM03] Daniel German and Audris Mockus. Automating the measurement of open source projects. In *Proceedings of ICSE '03 Workshop on Open Source Software Engineering*, Portland, Oregon, USA, May 2003.

[HF95] Jiawei Han and Yongjian Fu. Discovery of multiple-level association rules from large databases. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB 1995)*, pages 420–431, San Francisco, Ca., USA, September 1995. Morgan Kaufmann Publishers, Inc.

[HH03] Ahmed E. Hassan and Richard Holt. The chaos of software development. In *Proc. International Workshop on Principles of Software Evolution (IWPSE 2003)*, Helsinki, Finland, September 2003. IEEE Press.

[HK00] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan-Kaufmann Publishers, Inc., San Francisco, California, USA, 2000.

[HMS01] David Hand, Heikki Mannila, and Padhraic Smyth. *Principles of Data Mining*. MIT Press, Cambridge, Massachusetts, USA, 2001.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In Weidong Chen, Jeffery Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data: May 16–18, 2000, Dallas, Texas*, volume 29(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 1–12, New York, NY 10036, USA, 2000. ACM Press.

[HR83] T. Härder and A. Reuter. Principles of transaction oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983. Reprinted in [Sto88].

[IBM04] IBM. Eclipse innovation grant. 2004 award recipients. http://www-306.ibm.com/software/info/university/products/eclipse/eig-2004.html, January 2004.

[iSS04] Predicting Change Propagation in Software Systems. Ahmed e. hassan and richard c. holt. In *Proceedings of the International Conference on Software Maintenance (ICSM 2004)*, Chicago, Illinois, USA, September 2004.

[LFRGB04] Luis Lopez-Fernandez, Gregorio Robles, and Jesus M. Gonzalez-Barahona. Applying social network analysis to the information in CVS repositories. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, pages 101–105, Edinburgh, Scotland, UK, May 2004.

[MCK04] Frank McCarey, Mel Ó Cinnéide, and Nicholas Kushmerick. A case study on recommending reusable software components using collaborative filtering. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, pages 117–121, Edinburgh, Scotland, UK, May 2004.

[MFH02] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.

[Mic99] Amir 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, October 1999. IEEE Press.

[Mic00] Amir Michail. Data mining library reuse patterns using generalized association rules. In *International Conference on Software Engineering*, pages 167–176, 2000.

[Mic04] Microsoft Corporation. SQL server architecture – maximum capacity specifications. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/architec/8_ar_ts_8dbn.asp, January 2004.

[MM85] Webb Miller and Eugene W. Myers. A file comparison program. *Software—Practice and Experience*, 15(11):1025–1040, November 1985.

[MTV94] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Efficient algorithms for discovering association rules. In Usama M. Fayyad and Ramasamy Uthurusamy, editors, *AAAI Workshop on Knowledge Discovery in Databases (KDD-94)*, pages 181–192, July 1994.

[MV00] Audris Mockus and Lawrence 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, October 2000. IEEE.

[MWZ03] Audris Mockus, David M. Weiss, and Ping Zhang. Understanding and predicting effort in software projects. In *Proc. 25th International Conference on Software Engineering (ICSE)*, pages 274–284, Portland, Oregon, May 2003.

[Obj03] Object Technology International. *Eclipse Platform Technical Overview*, February 2003. Available at www.eclipse.org.

[OW04] Thomas J. Ostrand and Elaine J. Weyuker. A tool for mining defect-tracking systems to predict fault-prone files. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, pages 85–89, Edinburgh, Scotland, UK, May 2004.

[RD04] Filip Van Rysselberghe and Serge Demeyer. Mining version control systems for FACs (frequently applied changes). In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, pages 48–52, Edinburgh, Scotland, UK, May 2004.

[Rij79] C. J. Van Rijsbergen. *Information Retrieval, 2nd edition*. Butterworths, London, 1979.

[Riv92] Ron Rivest. The MD5 message-digest algorithm. Internet Request for Comment RFC 1321, Internet Engineering Task Force, April 1992.

[Riv01] Jim des Rivières. *How to use the Eclipse API*, May 2001. http://eclipse.org/articles/Article-API%20use/eclipse-api-usage-rules.html.

[SA95] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proceedings of the 21th Very Large Data Bases Conference (VLDB)*, pages 407–419, 1995.

[SBM98] Craig Silverstein, Sergey Brin, and Rajeev Motwani. Beyond market baskets: Generalizing association rules to dependence rules. *Data Mining and Knowledge Discovery*, 2(1):39–68, 1998.

[SGPP04] Kevin Schneider, Carl Gutwin, Reagan Penner, and David Paquette. Mining a software developer's local interaction history. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, pages 106–110, Edinburgh, Scotland, UK, May 2004.

[SM83] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, New York, 1983.

[Sof04] SoftChange. Project homepage. http://sourcechange.sourceforge.net/, June 2004.

[SSLM01] Jelber Sayyad-Shirabad, Timothy C. Lethbridge, and Stan Matwin. Supporting maintainance of legacy software with data mining techniques. In *Proc. International Conference on Software Maintenance (ICSM 2001)*, pages 22–31, Florence, Italy, November 2001. IEEE.

[SSLM03] Jelber Sayyad-Shirabad, Timothy C. Lethbridge, and Stan Matwin. Mining the maintenance history of a legacy software system. In *Proc. International Conference on Software Maintenance (ICSM 2003)*, Amsterdam, Netherlands, September 2003. IEEE.

[SSLM04] Jelber Sayyad-Shirabad, Timothy C. Lethbridge, and Stan Matwin. Mining the software change repository of a legacy telephony system. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, pages 53–57, Edinburgh, Scotland, UK, May 2004.

[Sto88] M. Stonebraker. *Readings in Database Systems*. Morgan-Kaufmann Publishers, Inc., San Francisco, California, USA, 1988.

[Sub04] Subversion. Project homepage. http://subversion.tigris.org/, June 2004.

[SVA97] 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, August 1997.

[Wag02] Mitch Wagner. Amazon.com admits concocting some recommendations. TechWeb News, http://www.techweb.com/wire/story/TWB20021204S0009, December 2002.

[Wei04] Peter Weißgerber. Design and application of an API for mining CVS repositories. Master's thesis, Saarland University, Saarbrücken, Germany, January 2004.

[WH04] Chadd Williams and Jeffrey K. Hollingsworth. Bug driven bug finders. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, pages 70–74, Edinburgh, Scotland, UK, May 2004.

[Yin03] Annie Tsui Tsui Ying. Predicting source code changes by mining revision history. Master's thesis, University of British Columbia, Canada, October 2003.

[YMNCC04] Annie T.T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining revision history. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, page 63, Edinburgh, Scotland, UK, May 2004.

[YR04] Yuhanis Yusof and Omer F. Rana. Template mining in source-code digital libraries. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, pages 122–126, Edinburgh, Scotland, UK, May 2004.

[ZDZ03] Thomas Zimmermann, Stephan Diehl, and Andreas Zeller. How history justifies system architecture (or not). In *Proc. International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 73–83, Helsinki, Finland, September 2003. IEEE Press.

[Zel03] Andreas Zeller. Program analysis: A hierarchy. In *Proceedings of ICSE 2003 Workshop on Dynamic Analysis (WODA 2003)*, pages 6–9, Portland, Oregon, May 2003.

[ZG03] Lijie Zou and Michael W. Godfrey. Detecting merging and splitting using origin analysis. In *Proc. 10th Working Conference on Reverse Engineering (WCRE 2003)*, Victoria, British Columbia, Canada, November 2003. IEEE.

[ZH02] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.

[ZW04] Thomas Zimmermann and Peter Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, pages 2–6, Edinburgh, Scotland, UK, May 2004.

[ZWDZ04] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proc. 26th International Conference on Software Engineering (ICSE)*, Edinburgh, Scotland, May 2004.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich diese Diplomarbeit selbständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle wörtlich oder sinngemäß übernommenen Ausführungen wurden als solche gekennzeichnet. Weiterhin erkläre ich, dass ich diese Arbeit in gleicher oder ähnlicher Form nicht bereits einer anderen Prüfungsbehörde vorgelegt habe.

Passau, den 1. Juni 2004

. . . .

Thomas Zimmermann