Adrian Bachmann
Abraham Bernstein

Data Retrieval, Processing and Linking for Software Process Data Analysis

TECHNICAL REPORT – No. IFI-2009.0003b

December 2009

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

A. Bachmann
A. Bernstein: Data Retrieval, Processing and Linking for Software Process Data Analysis
Technical Report No. IFI-2009.0003b, December 2009

Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL:

Data Retrieval, Processing and Linking for Software Process Data Analysis

Adrian Bachmann and Abraham Bernstein
Department of Informatics
University of Zurich
8050 Zurich, Switzerland
{bachmann,bernstein}@ifi.uzh.ch

Abstract—Many projects in the mining software repositories communities rely on software process data gathered from bug tracking databases and commit log files of version control systems. These data are then used to predict defects, gather insight into a project's life-cycle, and other tasks. In this technical report we introduce the software systems which hold such data. Furthermore, we present our approach for retrieving, processing and linking this data. Specifically, we first introduce the bug fixing process and the software products used which support this process. We then present a step by step guidance of our approach to retrieve, parse, convert and link the data sources. Additionally, we introduce an improved approach for linking the change log file with the bug tracking database. Doing that, we achieve a higher linking rate than with other approaches.

1 Introduction and Motivation

In general, software projects make use of a software developing process. In most cases this process is supported by tools such as an Integrated Development Environment (IDE), a software repository used as version control system, and a bug tracking database for maintenance purposes.

Because all of these systems store data during the process, they are valuable sources of information for the evolution and history of a software project / system. The combination of these information sources (e.g., the bug tracking database and the version control system) provides even more valuable information about the process history.

Modern software project management systems like Jazz[1] or Telelogic Synergy[2] provide full (or at least partially full) functionality needed to develop and maintain a software product in one single system (e.g., a version control system and a bug tracking database). Additionally, these systems often only allow a change on the source code in combination with a task, which might be a bug that should be fixed (i.e., a bug report), a new feature (i.e., existing feature request), or another task. As a result, the process data generated by these systems is well-integrated as far as these systems are used properly (e.g., no changes linked to empty work items).

Unfortunately, these systems — most of them are commercial products — are not widely-used in current software projects, and different systems are used to support the development and maintenance of a software product. However, the integration of data from these mostly stand-alone systems cannot be performed automatically, but has to be maintained manually by the developers. Conscientious developers, for instance, refer to a given bug report in the bug tracking database by typing the bug number in the commit message of the version control system. They usually do this without any formal guidelines on how to do so (or do not obey these).

Therefore, the integration between a version control system and a bug tracking database has to be established e.g., by scanning through the commit messages for valid bug report numbers (see [1]) which is indeed an inexact heuristic.

This integrated process data can be used e.g., to predict the location and number of future or hidden bugs, which is one of the challenges in current software engineering research (e.g., see [2], [3], [4], [5]). Project managers could use such predictions to identify the critical parts of a system, limit the gravity of their impact, and allow a better planning of testing and engineering efforts. These predictions therefore are valuable information. In other research areas such data is used, for instance, for software evolution visualization or process (data) quality analysis/measurement.

Presenting how to retrieve, process and link this process data for the previously announced application areas is one of the goals of this technical report. Our purpose is to get improve/enhance the quality of process data for further research, which base on such data.

We shortly introduce some of the most relevant tools

Technical Report. Dynamic and Distributed Information Systems Group, Department of Informatics, University of Zurich, Switzerland.
Published online in May, 2009.
http://www.ifi.uzh.ch/ddis/people/adrian-bachmann/pdq/

1. http://www-01.ibm.com/software/rational/jazz/
2. http://www.telelogic.com/products/synergy/

used in the software engineering process such as bug tracking databases and version control systems. Furthermore, we present a procedure for retrieving and processing the process data followed by introducing an adapted algorithm to integrate the version control system with the bug tracking database. Using our adapted algorithm, we achieve a better linking rate than in other data sets which have been presented and used for instance by Zimmermann et al. (see [6] or [3]). In the last part of this report we discuss the related and the future work.

2 Software Engineering Process Data

State-of-the-art software engineering models incorporate the following process steps (simplified):

- System Requirement Engineering
- Software Requirements Specification
- Software Design
- Software Implementation
- Software Integration; Formal Machine Testing
- Operation and Maintenance

Other development processes such as the prototyping approach, the spiral model, the iterative process etc. have different process steps but in all models we have at least an implementation and maintenance part which is more or less common. Therefore, the remainder part of this report focuses on these two process steps.
The fundament for software process data analysis is provided by data from systems in-process used in these two process steps. We will introduce these systems in detail in the following subsections.

2.1 Integrated Development Environments

Usually, a piece of software is coded in a Integrated Development Environment (IDE) which provides features such as syntax highlighting, in-program compiling, and debugging. Depending on the programming language, a number of IDEs are available[3]. But an IDE is only the environment for writing the code. Besides of the source code and its documentation, there is no more information available about the development process or the evolution of a project.
Thus, an IDE is no data provider for our research but can be used to provide enhanced functionality (based on process analysis) to the developer which allows a faster, better, or less defective coding in future. Hence, we do not discuss these systems further in this report.

2.2 Version Control Systems

As soon a project exceeds the number of one active developer, a version control system is needed to store the changes on the source code, and allowing concurring implementing activities. Such systems like the Concurrent Versions System (CVS)[4] or Subversion (SVN)[5] are widely-used in open as well as closed source projects and store the whole history of a software product including all changes.
Several developers may work on the same project concurrently, each one editing files within their own "working copy" of the project, and sending (or commit) their modifications to the version control system. If the commit operation succeeds (no conflicts), the version control systems updates all files involved, writes a user-supplied description line, the date, and the login name to its log files.
Based on this functionality, it is possible to obtain a change log file, which contains the following information for each commit / committed file:

- author,
- date and time,
- changed files,
- and an optional log message.

Because CVS and SVN are different in the way how they store the data (see Section 2.2.1), we use the following definition of a commit in the context of this report: "A commit refers to submitting the latest changes of the source code to the repository." Or in other words: a commit accords a transaction.
Because these change log files contain all the information about the history and evolution of a software project, they are a very valuable source of information for the development process.

2.2.1 *File Based (CVS) vs. Transactional Based (SVN)*

CVS and SVN are both widely used version control systems in open source software (OSS) projects as well in closed source software (CSS) projects and are therefore in many cases our data providers. Thus, it is important to know how common these two systems are, especially in the way how they store the version controlled data.
Both systems provide a state of the art functionality of a version control system, but they differ each other significantly in how they versioning the project repository data. Whereas CVS firstly addresses the data by location (L) and secondly by time (T), SVN goes the other way.
Summarized, we have the following data versioning techniques:

- CVS: (1) project, (2) location, (3) time → (P:L.T)
- SVN: (1) project, (2) time, (3) location → (P:T.L)

Based on the versioning technique, SVN handles the changes transaction oriented by building a new version/revision on the whole project with every commit (see Figure 1), whereas CVS has an independently version/revision on each single file (see Figure 2).

2.3 Bug Tracking Databases

Even a project team realizes a software product very carefully and spent much time on testing, the chance to have defects in the software product is nevertheless

3. See for instance http://en.wikipedia.org/wiki/Comparison_of_integrated_development_environments
4. http://www.nongnu.org/cvs/
5. http://subversion.tigris.org/

```
RCS file: /cvsroot/eclipse/org.eclipse.debug.ui/ui/org/eclipse/debug/internal/ui/views/memory/renderings/HexRendering.java,v
Working file: org.eclipse.debug.ui/ui/org/eclipse/debug/internal/ui/views/memory/renderings/HexRendering.java
head: 1.11
[...]
revision 1.10
date: 2007-01-20 00:10:46 +0100;  author: schan;  state: Exp;  lines: +2 -83;  commitid: ccb45b14ff64567;
Bug 114377:  [Memory View] Endian in hex view and ASCII view doesn't work
----------------------------
[...]
----------------------------
revision 1.1
date: 2005-02-09 18:32:56 +0100;  author: darin;  state: Exp;
Bug 84799 - Implement Memory View and renderings with new rendering APIs
=============================================================================
```

Fig. 2. CVS change log file (example of Eclipse IDE)

```
------------------------------------------------------------------------
r653772 | jim | 2008-05-06 15:38:00 +0200 (Die, 06 Mai 2008) | 2 lines
Geänderte Pfade:
   M /httpd/httpd/branches/2.2.x/STATUS
   M /httpd/httpd/branches/2.2.x/modules/ldap/util_ldap.c

PR: 44560

------------------------------------------------------------------------
r653770 | jim | 2008-05-06 15:37:07 +0200 (Die, 06 Mai 2008) | 2 lines
Geänderte Pfade:
   M /httpd/httpd/branches/2.2.x/CHANGES
   M /httpd/httpd/branches/2.2.x/STATUS
   M /httpd/httpd/branches/2.2.x/modules/proxy/mod_proxy_http.c

PR 44165

------------------------------------------------------------------------
```

Fig. 1. SVN change log file (verbose; non-XML; example of Apache HTTP Server)

high. Withrow showed in [7] for instance that the defect density[6] in modules written in Ada[7] lies in average between 0.5 and 1.9 - depending on the module size.

Such defects (or bugs) are normally reported in a bug tracking database, which supports the whole bug fixing process (as a part of the software maintenance processes).

For a better understanding of the data held in such bug tracking databases and its interpretation, we shortly introduce a classical bug fixing process followed by a deeper view to often used bug tracking databases.

2.3.1 Bug Fixing Process

Ideally, each software project has a well defined bug fixing process which is supported by a bug tracking database. Crowston (see [8]) defined the following main activities in a bug fixing process (actors in brackets):

- Find a problem while using system (Customer)
- Attempt to resolve problem (Response Center)
- Attempt to find workaround (Marketing Engineer)
- Diagnose the problem (Software Developer)
- Design a fix for the bug (Software Developer)
- Write the code for the fix (Software Developer)
- Recompile the module and link it with the rest of the system (Integrator)

If we focus more on the data part, we can describe the activities as follows (systems in brackets):

- Report a problem (Bug Tracking Database)
- Dispatch the problem-report to a developer (Bug Tracking Database)
- Check-out the current software version (Version Control System)
- Analyze and fix the problem (Integrated Development Environment)
- Check-in the fixed software version (Version Control System)
- Verify the fixed software version against the problem report (Bug Tracking Database)

Modern bug tracking databases support this process by providing a bug report status model. In the next subsection we describe common used bug tracking database systems in more detail.

2.3.2 Used Systems

Non-Commercial (”free”) open source bug tracking databases such as Bugzilla[8] or IssueZilla[9] are very popular in OSS projects, whereas in CSS projects likely commercial bug tracking and testing suites such as Quality Center[10] are used.

All these bug tracking databases support the bug fixing process with a status model, possibilities to discuss an issue, and track the fixing progress. As a common set of attributes, a bug report basically contain the following information:

- Bug ID (unique identification number)
- Summary and description of the bug
- Reporter name and/or email address
- Assignee name and/or email address
- Current status of the bug (e.g., New, Verified, Assigned, Closed, etc.)
- Resolution of the bug report (e.g., Fixed, Duplicate, etc.)
- Priority of the bug
- Date of reporting the bug
- Further product specific attributes such as operating system, hardware, web browser etc.

6. The defect density is defined as the number of defects per 1’000 lines of code (KLOC).
7. See e.g., http://www.adahome.com/ for more information to the Ada programming language.
8. http://www.bugzilla.org/
9. IssueZilla is no longer available for download.
10. https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24^1131_4000_100__

- Attachments such as screen-shots, stack-traces, etc.
- Bug report related comments (discussion)

Additionally, changes on the bug report are tracked in a so called ***activity log***.
Depending on the bug tracking database product, there are additional attributes such as severity or issue type.
Because, these systems store information to the whole bug fixing process (and sometimes about implementing new features), they are a very valuable source of information to analyze at least part of the software engineering process.

2.3.3 Public Available Bug Tracking Databases

Especially in open source projects, the bug reports maintained by bug tracking databases are open to everyone. In some cases a registration is needed, but there are typically no limitations on who is allowed to report a bug. Only security relevant bug reports are limited in their access to certain users.
Thus, the quality of these data may vary as the databases can contain spam bug reports, duplicates, and feature requests camouflaged as reported bugs (see [9] for more information about the quality of bug reports).

3 Data Retrieval, Processing and Linking

As the data sources are identified and well-known, we present in this section how to prepare the software engineering process data for further analysis purposes. We decided to store all the process data in a relational database system (e.g., MySQL[11]) which easily allows further analysis, because we can extract, combine and select the data in any desired format.
First, we describe the exact procedure to retrieve, parse and convert the data, and afterwards introduce our improved linking approach.

3.1 Data Retrieval

First, we have to fetch the data out of the original systems. To get the process data (change log) from version control systems, the procedure is straight-forward, if we have direct read access to the repository. The procedure for getting the process data from bug tracking databases requires more effort, depending on the access possibilities. In the optimum case, we can directly access to the underlying database and perform a database dump. Retrieve the bug data without database access, is more complicate. In this case, we have to fetch the data over the web interface followed by further data processing steps like parsing etc..
In the following subchapters, we describe these data retrieval methods in more detail.

11. http://www.mysql.com/

3.1.1 Version Control Systems

Version Control Systems such as CVS or SVN allow to get a change log file of all revisions/files by using one command. Additionally, SVN allows to retrieve the whole change log (including all changed files) in an XML format:

```
svn log --xml --verbose
```

Thus, we can simply export the change log data into a text or XML file. Note again, that there are differences in the change log files between CVS and SVN (see Section 2.2.1). We will cover the handling of these differences later in this section.

3.1.2 Bug Tracking Databases: Database Access

Having direct access to the underlying database system of a bug tracking database simplifies things: We can perform a database dump and fetch all the data needed. Unfortunately, in most cases we do not have direct access to the database system, unless we have an agreement with the operator of the bug tracking database (which is mostly the case in CSS projects).

3.1.3 Bug Tracking Databases: Web Interface

In OSS projects, the bug tracking databases are mainly available on the internet and readable to the public — often even without registration (see Section 2.3.3). Thus, we can fetch the bug data over the web user interface, although we do not have access to the underlying database system, which stores all the bug tracking information.
Luckily, Bugzilla and IssueZilla provide the bug report data, in addition to HTML, in an XML format. Thus, we can download a XML file of every bug report. To download all information, we used a wget[12] based shell script.
IssueZilla provides all the information including the activity log in these XML files. Bugzilla, depending on the version, does not include the activity log in the XML file. Thus, we have to download the activity log separately as HTML file.
Finally, we have one XML file and optionally one HTML file for each bug report.

3.2 Data Parsing

After retrieving the data files, the raw data has to be parsed into a relational database. To do this job, we have written several XML and HTML parsers.
In the following subsections, we describe the procedure to parse the data in more detail.

3.2.1 Version Control Systems: Change Log File

As already mentioned in Section 3.1.1, SVN provides the change log file optionally in an XML format, which can be parsed quite easily with an XML parser/parser

12. http://www.gnu.org/software/wget/

library[13] or any XML editor.
In contrast, CVS does not provide the change log file in an XML file format. Thus, we wrote a simple text parser which extracts all the relevant attributes for each revision out of the CVS change log file (Figure 2 shows an extract of a CVS change log file).
Note again, that the CVS and SVN change log file do not contain the same level of information (see Section 2.2.1). The CVS change log is file based whereas the SVN change log file is transactional oriented.

3.2.2 Bug Tracking Databases: Database Dump

In case we were able to extract the bug tracking data directly from a database system, we can load this database dump directly back into our relational database without any additional parsing effort.

3.2.3 Bug Tracking Databases: Bug Report Files

Having the data stored in XML and HTML files, needs additional effort, whereas parsing these files is quite similar to the version control system change log file. We wrote a parsing tool, which runs through all the files and uses the SAX XML parser to get the needed data out of the XML files. The HTML files were parsed with simple text parsing techniques, again (similar to the CVS change log file). Because the Bugzilla activity log is stored in a HTML table, the text parsing algorithm is simple.

3.3 Data Conversion

For our research, we are interested in a process oriented view on the data of the software engineering process. Therefore, we need to obtain a transaction-oriented view of the data, that maintains its temporal flow. By transaction we mean the concept, what information was produced by whom in a single commit (see Section 2.2). Fortunately, bug tracking databases such as Bugzilla or IssueZilla provide the data already in this form. Analyzing the change logs can be a bit more involved.
SVN has a transactional change log file (see Section 2.2 and Figure 1) per default, which only needs to be condensed. CVS, in contrast, maintains a file-level change log, which allows not a transactional view on the data by default (Figure 2 shows a small extract of a file-based CVS change log file). Thus, we have to reconstruct a transactional change log file to get a process oriented view on the data.
The reconstruction of a CVS change log file into a transactional change log file is not a huge effort, because the developer typically checked-in all changed files into the repository with one single commit. Therefore, the developer and the commit message of one transaction (commit) are the same (see [10]). The date and time information can be (slightly) different, as every single file is uploaded separately and gets therefore its own time stamp. If a developer commits two large files, for example, the files may not have the same commit time information in the change log entries.
Consequently, we take the complete CVS change log file of a project and sort the entries by author, commit message and date/time. Next, we combine the change log elements with the same author and commit message and a given maximum time difference to one transaction (sliding window approach) similar to a SVN change log file (see Figure 3). A maximum time difference (time window) of up to five seconds between the change log entries turned out to be an optimal value. Finally, we assign a transaction number to all reconstructed transactions ordered by date and time to get an unique identification number for each transaction/commit.

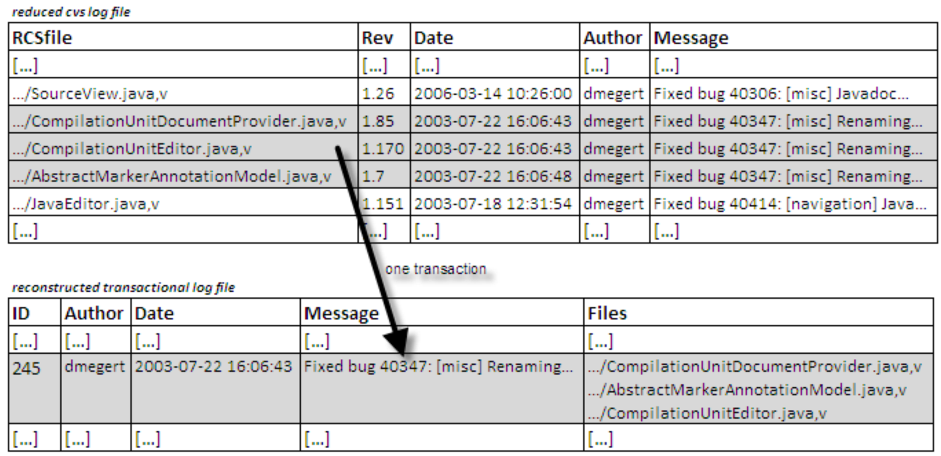
3.4 Linking the Data Sources

The analysis of process data requires linking the version control system information with the bug tracking information. Modern software project management systems like Jazz or Telelogic Synergy only allow a commit in combination with a task, which can be a bug that should be fixed (i.e., a bug report), a new feature (i.e., existing feature request), or another task. As a result, the data generated by these systems is already well-linked.
Unfortunately, these systems are not widely-used in current software projects, however, this linking is not established automatically but has to be manually maintained by the developers. Conscientious developers refer to a given bug report by typing the bug number in the commit message. They mostly do this without any formal guidelines on how to do so (or there are guidelines, but nobody keeps them).
We establish the links between the version control system information with the bug tracking database information in a procedure comparable to Fischer et al. [1] by scanning trough the log messages for potential bug report IDs. Additionally, we improved this approach to get a better linking rate and verified links at the same time.
In the following subsections we define what fixed bug reports are and introduce our improved linking approach. We close this section with a view to the censored data issue.

3.4.1 Fixed Bug Reports

With our improved linking approach, we verify the potential bug report links based on the chance of its correctness. As already mentioned in Section 2.3.3, the bug tracking database contains not only fixed bug reports, for which a bug fix could be implemented, rather than duplicates, feature requests, etc..
We assume that only numbers in the change log which refer to resolved, verified or closed bug reports, which have a reported fixing activity, could be valid. We call these bug reports “fixed” bug reports with the following definition: ***Fixed bug reports*** are bug reports that have at least one associated fixing activity within the considered time period.

13. e.g., http://www.saxproject.org/



reduced cvs log file

RCSfile	Rev	Date	Author	Message
[...]	[...]	[...]	[...]	[...]
.../SourceView.java,v	1.26	2006-03-14 10:26:00	dmegert	Fixed bug 40306: [misc] Javadoc...
.../CompilationUnitDocumentProvider.java,v	1.85	2003-07-22 16:06:43	dmegert	Fixed bug 40347: [misc] Renaming...
.../CompilationUnitEditor.java,v	1.170	2003-07-22 16:06:43	dmegert	Fixed bug 40347: [misc] Renaming...
.../AbstractMarkerAnnotationModel.java,v	1.7	2003-07-22 16:06:48	dmegert	Fixed bug 40347: [misc] Renaming...
.../JavaEditor.java,v	1.151	2003-07-18 12:31:54	dmegert	Fixed bug 40414: [navigation] Java...
[...]	[...]	[...]	[...]	[...]

reconstructed transactional log file

ID	Author	Date	Message	Files
[...]	[...]	[...]	[...]	[...]
245	dmegert	2003-07-22 16:06:43	Fixed bug 40347: [misc] Renaming...	.../CompilationUnitDocumentProvider.java,v .../AbstractMarkerAnnotationModel.java,v .../CompilationUnitEditor.java,v
[...]	[...]	[...]	[...]	[...]

Fig. 3. Reconstruction of the transactional change log file (simplified example of Eclipse IDE)

3.4.2 Improved Linking Approach

To establish the links between a commit/transaction and a bug report, we have to scan through the commit/transaction messages for valid bug report numbers (see [1]) which is an inexact heuristic. To improve this process, we adapted the original algorithm (1) and built in verification steps (2-4):

1) Scan through the commit/transaction messages for numbers in a given format (e.g., "PR: 112233"), or numbers in combination with a given set of keywords (e.g., "fixed", "bug", etc.).
2) Exclude all false-positive numbers (e.g., release numbers, year dates, etc.), which have a defined format.
3) Check, if the potential bug number exists in the bug database.
4) Check, if the linked bug report is a fixed bug report and whether it has a fixing activity 7 days before or 7 days after the commit date (see Figure 4).

The process tries to match numbers used in commit/transaction messages with bug numbers. For all positive matches it then established if the corresponding bug was fixed in the period of 7 days before or 7 days after the relevant commit – a time period that seemed optimal for the projects we investigated (see Section 3.4.3).
Our linking approach differs from previously used approaches (e.g., [6], [3], [1], [11]) in that it is less restricted to potential numbers but verifies them in the steps 3 and 4.

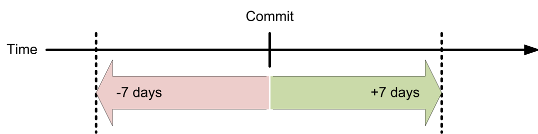


Fig. 4. Valid time period for bug report links (existence of a fixing activity in this time period)

3.4.3 Bug Report Links: Valid or Not?

A difficult question in this context is the following: In which cases ia a link valid and in which cases not?
To discuss this issue, first we recall the bug fixing process (see Section 2.3.1). Assuming we have a bug tracking database such as Bugzilla or IssueZilla and a version control system such as CVS or SVN. A developer would use these systems in the bug fixing process as follows:

1) Read the bug report and analyze the problem (status = assigned)
2) Get the current version of source code, fix the problem and commit the changed version by declaring the bug report id in the commit message
3) Change the status of the bug report to resolved (status = resolved; resolution = fixed)

To check the validity of a link, we now use the time difference between step 2 and 3. Normally, a developer commits the fixed version of the source code and minutes to hours later, he also changes the status of the corresponding bug report. If we scan trough the change log and get links to fixed bugs which were marked as fixed e.g., 180 days after the commit, we have to assume, that this is not a valid link. It is possible that the developer misspelled the number or the number is simply no bug report id in this context (false-positive bug ID).
On the other hand, developers tend to mark a bug as fixed and commit more source code changed hours to days later, which are linked to the already "fixed" bug. Particularly in CSS projects this is a common practice due to deadline/service level agreement restrictions.
Thus, the definition of a time period for valid links is always a trade off between more links (less false-negatives) and a higher chance of invalid links (more false-positives). We can check even whether a linked bug report exists and the link makes sense, we are never sure, if this linking is really valid or not. In other words: We do not know if the source code change which was committed to fix a reported bug, really fixed the problem

TABLE 1
Time Difference between Commit and Bug Report Status Change (Eclipse)

Time diff (days)	Proportion of data
<-7d	0.11%
-7d	0.21%
-6d	0.21%
-5d	0.17%
-4d	0.26%
-3d	0.41%
-2d	0.38%
-1d	1.54%
0d	74.36%
1d	4.42%
2d	1.62%
3d	1.61%
4d	1.24%
5d	1.14%
6d	1.04%
7d	0.85%
8d	0.62%
9d	0.37%
10d	0.37%
11d	0.39%
12d	0.30%
13d	0.35%
14d	0.56%
15d	0.40%
16d	0.20%
17d	0.21%
18d	0.20%
19d	0.18%
20d	0.22%
>20d	6.07%

which was reported in the linked bug report. The only way to do that, is source code inspection for every bug report, which needs plenty of time.
To get an optimal value for the valid time period, we analyzed the time difference between the commit and the status change (resolution=fixed) in the bug report without any time constraint (leave step 4 presented in Section 3.4.2).
The results for the Eclipse project are shown in Table 1. Defining a valid time window of ±7 days (see Figure 4), we considered almost 92% of all potential bug report links. Almost 75% of bug report links have a status change on the bug tracking database prior 24 hours (0d) after the commit. A verification by manual inspection of the data for false-positives or false-negatives approved this time range.
Similar results could be obtained for other — OSS and CSS — projects.

3.4.4 Censored Data

One important issue to keep in mind when discussing fixed bugs or verified links is censored data. Every data set that only uses a subset of the overall original data or uses continuous data will have missing information due to incomplete data (famous boundary problem). For example, if we employ a data set ranging from 2008-01-01 to 2008-12-31 for all entries and look at a bug report entered on December 30th, 2008 then a commit by a developer on December 31th, 2008 and bug status change to "fixed" on January 1st, 2009 will appear to be a data quality issue: In the above example data set we do not know anything about the commit and the status change and would, therefore, assume that the bug report link is wrong, due to missing fixing activity. Whilst this problem may appear artificial, we will always have to consider the boundaries.
To limit the error introduced by this censoring, we should purposefully employ data sets including very long periods of time (many years).

4 EVALUATION

Many research areas such as bug prediction, software visualisation, process quality measurement/analysis etc. need to have proper software engineering process data. Otherwise the respecting results could be false, misleading or incomplete.
In this technical report, we presented an improved approach to retrieve, process and link this software engineering process data. But how qualitative is this data?
To get an indication about that, we checked the quality of our resulting process data in different ways.
For that reason, we gathered the data of five CSS projects and one OSS project:

- Apache HTTP Server[14]
- Eclipse IDE[15]
- GNOME Desktop Project[16]
- NetBeans IDE[17]
- OpenOffice productivity suite[18]
- Banking System (CSS)[19]

14. http://httpd.apache.org/
15. http://www.eclipse.org/
16. http://www.gnome.org/
17. http://www.netbeans.org/
18. http://www.openoffice.org/
19. Due to security and confidentiality considerations we are not allowed to publish more information about this project.

4.1 Retrieving, Parsing and Conversion Quality

Our approach to retrieve, parse and convert the data is quite similar to previous approaches (see Chapter 5). Thus, we focused the quality assurance on double checking the original data with the parsed data. Since we have no manipulation on the data, we can simply compare the original with the parsed and converted data. After checking our procedures and algorithms without any problem issues, we assume to have no quality issue in this point.

4.2 Linking Quality

Checking the quality of the links between the change log and the bug tracking database can be a bit more involved.

Our linking approach differs from previously used approaches (e.g., [6], [3], [1], [11]) in that it is less restricted to potential bug reference numbers but verifies them (see Chapter 3.4.2). The result is a better linking rate (e.g., 43.7% compared to 24.3% for Eclipse[20]).

But is this better linking rate a result of a higher false-positive rate and, what about false-negatives?

We checked our data sets by manual inspection for false-positives e.g., whether there are identified bug report numbers which can not be truly a bug report link. Finally, we got a false-positive rate for all data sets which is far below 1%. Thus, our better linking rate is not a result of a much higher-false positive rate.

Additionally, we checked the data sets for false-negatives which is more difficult (see Chapter 3.4.3). Keep our restriction of ±7 days for valid bug report links, we got a false-negative rate which is far below 1% (e.g., bug report numbers which are written with separators e.g., "Bug #223'344 fixed" or "Bug #22 33-44 fixed").

4.3 Overall Quality

Based on our manual inspection results on false-positives and false-negatives, we assume that we find virtually all bug report links which the developers established by linking a bug report numbers in the commit message.

Nevertheless, the linking rate in all analyzed data sets (except of Apache HTTP Server) is far below 50%. This low rate is not a result of a poor linking algorithm, but a result of missing bug report link information in the change log. Unfortunately, these very low rates can influence the results of research/applications which base on such process data. We addressed this problem already in [12].

In our future work, we plan to analyze this impact and the general quality of software engineering process data more in detail (see Chapter 6).

20. Based on the data set provided on http://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/

5 RELATED WORK

Mining software engineering process data (also known as mining software repositories) got very popular in the last few years. Thus, much literature exists about this topic.

First of all, analyzing these process data requires the knowledge of the bug fixing process. Koru and Tian [13] give a good introduction to the bug fixing process. At the same time, they analyzed the details of this process in several OSS projects.

Fischer et al. [1] presented a Release History Database (RHDB) which contains the CVS change log and the bug report information (excluding the activity log) of all bug reports which are linked in the change log. To link the change log and the bug tracking database, Fischer et al. searched for change log messages which match to a given regular expression (e.g., `bugi?d?:?=?\s*#?\s*(\d\d\d+)(.*)` or `b=(\d\d\d\d+)(.*)`. In this first version of the algorithm, no verification step was done. Later, they improved the algorithm by checking for changed files or modules in the bug tracking database [14].

A similar approach to link the change log with the bug tracking database was chosen by other researchers. All of them used regular expressions to find bug report link candidates in the change log file. Čubranić and Gail [11] verified the link candidates by checking whether any activity occurred on the linked bug report within a small time frame around the commit, which is quite similar to our approach. Śliwerski et al. [3] verified the candidates with a semantic analysis (e.g., the linked bug report has been resolved as fixed at least once) whereas Schröter et al. [15] and Zimmermann et al. [6] checked whether a candidate log message contains keywords such as "fixed" or "bug" or matches patterns like "# and a number".

Finally, a similar approach to reconstruct the transactional change log file from a CVS change log was presented by Zimmermann and Weissgerber [10].

6 FUTURE WORK

Our work provides approved and enhanced software engineering process data, which can be used for applications such as defect prediction, software engineering quality analysis etc. which rely on such process data.

In order to make our findings actionable, we intend to investigate what factors influence bad software process data quality, and if the data contains any systematic bias (see [12]).

We would also like to further analyze the quality of the software engineering process data. Thus, we plan to define several data quality and project measures which specify the (data) characteristics of a software project. With such measures, we should be able to better understand and classify the data. This could also help to parameterize future applications project specific.

Furthermore, we plan to provide a case study based

on these quality and characteristics measures to get a comprehensive view on the data quality and the project characteristics of several OSS and CSS projects.

7 CONCLUSION

In the first part of this paper we introduced some details to the software engineering process including the bug fixing process in detail. Additionally, we showed in which software systems process data is held.

Later, we presented our approach for data retrieval, processing and linking in detail. Particularly we showed our approach to reconstruct the transactional change log from CVS change log data and introduced our adapted algorithm to link the change log data with the bug tracking database data. At the same time, we dealt with the question, which bug report links are valid and which are not.

In the evaluation part, we could successfully show the better linking rate of our approach compared to other linking approaches. We also showed just how badly bug reports are linked in commit log messages and pointed to the problem of this poor data quality.

In summary, we presented an approach to retrieve, process and link software engineering process data on a approved high level of quality. The resulting process data can be used for further analysis and applications which base on such data.

Most important, even with our adapted linking approach, only a fraction of the fixed bugs do have links to the change log file. Thus, our analysis provided a basis for future work to draw upon the consequences other application areas might have when relying on such software process data.

ACKNOWLEDGMENTS

This work was partly funded by the Zurich Cantonal Bank. Many thanks to Tom Scharrenbach and other DDIS group members for reviewing this technical report.

REFERENCES

[1] M. Fischer, M. Pinzger, and H. C. Gall, "Populating a release history database from version control and bug tracking systems," in *Proceedings of the International Conference on Software Maintenance*. Amsterdam, Netherlands: IEEE Computer Society Press, September 2003, pp. 23–32.

[2] A. Bernstein, J. Ekanayake, and M. Pinzger, "Improving defect prediction using temporal features and non linear models," in *Proceedings of the International Workshop on Principles of Software Evolution*. Dubrovnik, Croatia: IEEE Computer Society Press, September 2007, pp. 11–18.

[3] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*. New York, NY, USA: ACM, 2005, pp. 1–5.

[4] H. Joshi, C. Zhang, S. Ramaswamy, and C. Bayrak, "Local and global recency weighting approach to bug prediction," in *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*. Washington, DC, USA: IEEE Computer Society, 2007, p. 33.

[5] M.-T. J. Ostrand, F.-E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 340–355, 2005.

[6] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, p. 9.

[7] C. Withrow, "Error density and size in ada software," *IEEE Softw.*, vol. 7, no. 1, pp. 26–30, 1990.

[8] K. Crowston, "A coordination theory approach to organizational process design," *Organization Science*, vol. 8, no. 2, pp. 157–175, 1997.

[9] N. Bettenburg, S. Just, A. Schroeter, C. Weiss, R. Premraj, and T. Zimmermann, "Quality of bug reports in eclipse," in *In Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange (eTX 2007)*, Montreal, Quebec, Canada, October 2007, pp. 21–25.

[10] T. Zimmermann and P. Weissgerber, "Preprocessing cvs data for fine-grained analysis," in *MSR '04: Proceeding of the 1st International Workshop on Mining Software Repositories (MSR)*, Edinburgh, UK, 2004, p. 5.

[11] D. Čubranić and G. C. Murphy, "Hipikat: recommending pertinent software development artifacts," in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 408–418.

[12] E. Aune, A. Bachmann, A. Bernstein, C. Bird, and P. Devanbu, "Looking back on prediction: A retrospective evaluation of bug-prediction techniques," Student Research Forum at SIGSOFT 2008/FSE 16, November 2008.

[13] A. G. Koru and J. Tian, "Defect handling in medium and large open source projects," *IEEE Softw.*, vol. 21, no. 4, pp. 54–61, 2004.

[14] M. Fischer, M. Pinzger, and H. Gall, "Analyzing and relating bug report data for feature tracking," in *Proceedings of the 10th Working Conference on Reverse Engineering*. Victoria, B.C., Canada: IEEE Computer Society Press, November 2003, pp. 90–99.

[15] A. Schröter, T. Zimmermann, R. Premraj, and A. Zeller, "If your bug database could talk..." in *Proceedings of the 5th International Symposium on Empirical Software Engineering. Volume II: Short Papers and Posters*, Rio de Janeiro, Brazil, September 2006, pp. 18–20.