Populating a Release History Database from Version Control and Bug Tracking Systems*

Michael Fischer, Martin Pinzger, and Harald Gall
Distributed Systems Group, Vienna University of Technology
{fischer,pinzger,gall}@infosys.tuwien.ac.at

Abstract

Version control and bug tracking systems contain large amounts of historical information that can give deep insight into the evolution of a software project. Unfortunately, these systems provide only insufficient support for a detailed analysis of software evolution aspects. We address this problem and introduce an approach for populating a release history database that combines version data with bug tracking data and adds missing data not covered by version control systems such as merge points. Then simple queries can be applied to the structured data to obtain meaningful views showing the evolution of a software project. Such views enable more accurate reasoning of evolutionary aspects and facilitate the anticipation of software evolution. We demonstrate our approach on the large Open Source project Mozilla that offers great opportunities to compare results and validate our approach.

1. Introduction

During the lifetime of a software project a large amount of historical information is collected and stored by version control systems such as CVS [10], ClearCase [4], or SourceSafe [2]. Since this information describes interesting aspects of the evolutionary changes of a project, they are a valuable source for retrospective analysis techniques which explore, for example, change rates (number of changes within a certain amount of time), error proneness (number of errors), or indicate starvation (number of changes converges to zero) of software projects.

Version information may be enhanced with data from bug tracking systems that report about past maintenance activities. Both information sources together lead to an extensive database that enables reasoning about the past and anticipating future evolution of software projects. Unfortunately, current version and bug report systems provide no or only insufficient support for the combination of both data sources and, hence, lack capabilities in software evolution analysis. Moreover, the formats of and access to version and bug report data vary across version control and bug report systems that complicates the application of evolution analysis techniques and tools.

In this paper we introduce the population of a *Release History Database* that combines version and bug report data and we further demonstrate some query examples with respect to software evolution analysis. The basic building blocks of our approach are an SQL database and scripts for retrieval and filtering information from (a) the version control system and (b) the bug report database. In particular, we demonstrate our approach on the Open Source project *Mozilla* [3] that uses CVS as version control system and *Bugzilla* [1] as bug report database. The *Mozilla* project also has been addressed by other approaches in the software evolution area [15, 20] and hence offers great opportunities to compare results and emphasize the advantages of our approach.

The remainder of this paper is organized as follows: Section 2 gives an overview about related work in the area of software evolution analysis. Section 3 describes the principles of CVS and *Bugzilla*. In Section 4 we address the problem of merge detection for development branches. Section 5 explains the data model which will be used in later case studies and Section 6 explains the import process. An evaluation of our release history database is presented in Section 7. We conclude in Section 8 with a discussion and future work which will be built upon data provided by the Release History Database.

2. Related work

To our knowledge there has not been much work in combining version control and bug report data for software evolution analysis. Most works focus on evolution aspects of software or their visualization.

*This work is partially funded by the Austrian Ministry for Infrastructure, Innovation and Technology (BMVIT) and the European Commission under EUREKA 2023/ITEA-ip00004 'from Concept to Application in system-Family Engineering (CAFÉ)'.

Similar to an earlier approach of our group described in [12] Kemerer and Slaughter use modification reports as basis for their analysis [17]. They use a refined classification scheme for the modification reports (Corrective, Adaptive, Perfective Enhancement [23], New Program) for an analysis of ordered change events. The focus of their work is on deducing mappings of the systems life cycle (new program, corrective, adaptive, etc.) by accomplishing gamma analysis followed by a mapping phase.

Ball et al. [8] focused on the visualization of statistical data derived from the version control system. A systematic approach to integrate information from external sources is not presented.

In [12, 13, 14] our group examined the structure of several releases of a large Telecommunications Switching Software (TSS). Since the release data were stored in an object oriented database this approach could not be reused for the investigation of other software systems using different version control systems.

3. CVS and Bugzilla

In this section we give an overview about the CVS version control system and the *Bugzilla* bug tracking system that are the two major data sources required by our evolution analysis of the *Mozilla* project.

3.1. CVS

Basically, CVS is designed to handle revisions of textual information by storing delta's between subsequent revisions in the repository. Binary files can be stored in the repository as well, but they are not handled efficiently.

Revision numbers: Typically, version control systems distinguish between version numbers of files and software products. Concerning files these numbers are called *revision numbers* and indicate different versions of a file. In terms of software products they are called *release numbers* and indicate the releases of a software product.

Each new version of a file stored in the CVS repository receives a unique revision number (e.g. `1.1` for the first version of a file checked in). After an update of a file and a commit of the changes to the CVS repository the revision number of each affected file is increased by one. Because some files are more affected by changes than others these files have different revision numbers in the CVS repository.

A release represents a snapshot on the CVS repository comprising all files realizing a software system whereas the files can have individual revision numbers. Whenever a new version of the software system is released a symbolic name (i.e. tag) indicating the release is assigned to the revision numbers of current files. The relation *symbolic name - revision number* is stored in the header section of every tagged file and also appears in the header section of CVS log files.

Branches are common to version control systems and indicate a self-maintained line of development [10]. Each branch is identified by its *branch number*. CVS creates a new branch number by picking the first unused even integer, starting with 2, and appending it to the file's revision number where the corresponding branch forked off. For example the first branch created at revision `1.2` of a file receives the branch number `1.2.2` (internally CVS stores `1.2.0.2`). The main issue with branches is the detection of merges that is not supported by CVS. We will come back to this problem in Section 4 where we describe an algorithm to identify merge points.

Version control data: For each working file in the repository CVS generates version control data and stores it to log files. From there, log file information can be retrieved by issuing the `cvs log` command. The specification of additional parameters allow the retrieval of information about a particular file or a complete directory. Figure 1 depicts an example log file taken from the *Mozilla* project showing version data of the source file `nsCSSFrameConstructor.cpp` as it is stored by CVS.

Basically, a log file consists of several sections, each describing the version history of an artefact (i.e. file) of the source tree. Sections are separated by a line of '=' characters. For the population of our release history database we take into account the following attributes:

RCS file: The path information in this field identifies the artefact in the CVS repository.

symbolic names: Lists the assignment of revision numbers to tag names. This assignment is individual for each artefact since revision numbers may differ.

description: Lists the *modification reports* describing the change history of the artefact starting from its initial check in until the current release. Besides the modifications made in the main trunk all changes which happened in the branches are also recorded there. Reports (i.e. revisions) are separated by a number of '-' characters. The *revision* number identifies the source code revision (main trunk, branch) which has been modified. Date and time of the check in are recorded in the *date* field. The *author* field identifies the person who did the check in. The value of the *state* field determines the state of the artefact and usually takes one of the following values: "Exp" means experimental and "dead" means that the file has been removed. The *lines* fields counts the lines added and/or deleted of the newly checked in revision compared with the previous version of a file. If the current revision is also a branch point, a list of branches derived from this revision is

Proceedings of the International Conference on Software Maintenance (ICSM'03)
1063-6773/03 $17.00 © 2003 IEEE

```
RCS file: /cvsroot/mozilla/layout/html/style/src/nsCSSFrameConstructor.cpp,v
Working file: nsCSSFrameConstructor.cpp
head: 1.804
branch:
locks: strict
access list:
symbolic names:
        MOZILLA_1_3a_RELEASE: 1.800
        NETSCAPE_7_01_RTM_RELEASE: 1.727.2.17
        PHOENIX_0_5_RELEASE: 1.800
        ...
        RDF_19990305_BASE: 1.46
        RDF_19990305_BRANCH: 1.46.0.2
keyword substitution: kv
total revisions: 976;   selected revisions: 976
description:
----------------------------
revision 1.804
date: 2002/12/13 20:13:16;  author: doe@netscape.com;  state: Exp;  lines: +15 -47
Don't set NS_BLOCK_SPACE_MGR and NS_BLOCK_WRAP_SIZE on ...
----------------------------
...
----------------------------
revision 1.638
date: 2001/09/29 02:20:52;  author: doe@netscape.com;  state: Exp;  lines: +14 -4
branches:  1.638.4;
bug 94341 keep a separate pseudo frame list for a new pseudo block or inline frame ...
----------------------------
....
=============================================================================

RCS file: /cvsroot/mozilla/layout/html/style/src/nsCSSFrameConstructor.h,v
```

Figure 1. Example log-file from Mozilla source tree

---

listed in the *branches* field (e.g. 1.638.4 in Figure 1). The following *free text* field contains informal data entered by the *author* during the check in process.

3.2. Bugzilla

As additional source of information to the modification reports, bug report data from the *Bugzilla* bug report database is imported into our Release History Database. Access to the *Bugzilla* database is enabled via HTTP and reports can be retrieved in XML format. The information will be used later to classify the corresponding modification reports found in CVS. This enables the identification of error-prone files or modules which are candidates for re-implementation or re-design.

Besides some administrative information such as contact information, mailing addresses, discussion, etc., the bug report database also provides some interesting information for the evolutionary view such as bug severity, affected product or component (see Figure 2):

bug_id: This ID is referenced in modification reports. Since the IDs are stored as free text in the CVS repository, the information cannot be reliably recovered from the change report database.

bug_status (status whiteboard): Describes the current state of the bug and can be *unconfirmed*, *assigned*, *resolved*, etc.

product: Determines the product which is affected by a bug. Examples in *Mozilla* are Browser, MailNews, NSPR, Phoenix, Chimera, etc.

component: Determines which component is affected by a bug. Examples for components in *Mozilla* are Java, JavaScript, Networking, Layout, etc.

dependson: Declares which other bugs have to be fixed first, before this bug can be fixed.

blocks: List of bugs which are blocked by this bug.

Proceedings of the International Conference on Software Maintenance (ICSM'03)
1063-6773/03 $17.00 © 2003 IEEE

```
<bug_id>100069</bug_id>
<bug_status>VERIFIED</bug_status>
<product>Browser</product>
<priority>--</priority>
<version>other</version>
<rep_platform>All</rep_platform>
<assigned_to>doe@mozilla.org</assigned_to>
<delta_ts>20020116205154</delta_ts>
<component>Printing: Xprint</component>
<reporter>doe@mozilla.org</reporter>
<target_milestone>mozilla0.9.6</target_milestone>
<bug_severity>enhancement</bug_severity>
<creation_ts>2001-09-17 08:56</creation_ts>
<qa_contact>doe@mozilla.org</qa_contact>
<op_sys>Linux</op_sys>
<resolution>FIXED</resolution>
<short_desc>Need infrastructure for new print
    dialog</short_desc>
<keywords>patch, review</keywords>
<dependson>106372</dependson>
<blocks>84947</blocks>
<long_desc>
 <who>doe@mozilla.org</who>
 <bug_when>2001-09-17 08:56:29</bug_when>
 <thetext></thetext>
</long_desc>
```

Figure 2. Snippet from Bugzilla file

bug_severity: This classification field for a bug report (blocker, critical, major, minor, trivial, enhancement)

target_milestone: Possible target version when changes should be merged into the main trunk.

4. Tracing evolution across branches

For the evolutionary analysis of Software Product Lines (but not limited to) it is desirable to trace back the introduction of new code, e.g., code of new features, in the main trunk back to its origins which also can be somewhere in a branch (see Figure 3).

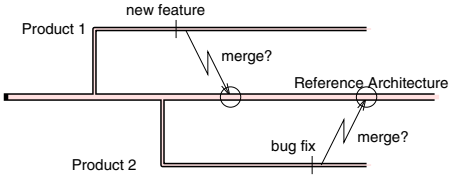

Figure 3. Features and products

Another motivation is given by the fact that modifications, e.g., bug fixes, can be applied first on a branch and later merged into the main trunk. In this case modification reports of the branch contain information which does not slip into the modification report of the main trunk during a merge.

In CVS merges are performed on a pure textual level and can be performed automatically only if no conflicting situation during the merge operation occurs. A merge is performed by first building the delta between two revisions followed by a modification phase whereas the differences are incorporated into the target file. A conflict situation arises when simultaneous changes within a single file in a mutual exclusive way are made to the same code section, e.g, a source line has been edited in one version and the same line has been deleted in a second version of the file. CVS does not resolve such conflicts, instead the affected text section is marked and the developer has to resolve the conflict manually. Usually this can be done without too much effort. When the situation is more difficult, the process of resolving the conflict requires communication between the responsible persons. Solutions for improvements of the merge process, such as the application of 3-way diff to build the delta, are discussed in [7, 19, 21]. Unfortunately, CVS does not provide mechanisms to record such merges.

To overcome this limitation, we developed an algorithm which delivers criteria for the acceptance or rejection of the hypothesis that source text found in a branch b also appears in in the main trunk m. Input to this algorithm are the revision tree data (revision numbers, revision date) and the differences between revisions of a source file.

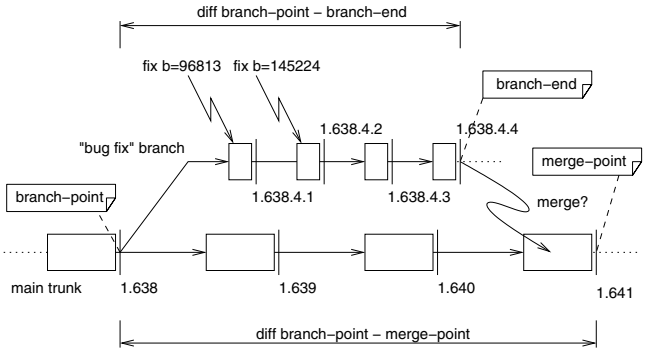

Figure 4. Example for branches/merges

Branch/merge algorithm: The algorithm is based on the observation that merges are performed either automatically through CVS and lines are copied into the main trunk or done by hand also by copying the source lines from the branch into the main trunk.

In the description of the algorithm we will use the following naming convention: a *branch-point* denotes a point in time and r_{bp} the associated revision number where the branch has been split of (see Figure 4); the last revision of the branch, i.e., the end of a branch, checked in into the repository is denoted by *branch-end* and r_{be}, respectively; *merge-point* and r_{mp} denote a possible candidate or the actual point where modifications of the branch have been merged into the main trunk and have been made public by

Proceedings of the International Conference on Software Maintenance (ICSM'03)
1063-6773/03 $17.00 © 2003 IEEE

updating the repository; S_{be} denotes the set of new lines introduced to the branch form the *branch point* until the *branch-end*; S_{mp} denotes the set of new lines in the main trunk. The outline of the algorithm is as follows:

1. Get the set of of source lines, i.e., S_{be}, introduced between r_{bp} and r_{be} from the repository using the `cvs diff` command (or alternatively from the local files if the releases are available);
2. Determine a possible *merge-point* (r_{mp}) by retrieving the last revision date of the *branch-end* (r_{be}) and find a revision date in the main trunk which is later than that of the branch, i.e., a *merge-point* candidate;
3. Get a second "diff" to obtain the differences between r_{bp} and r_{mp}, i.e., S_{mp};
4. Select in both sets S_{be} and S_{mp} the lines which have been added since r_{bp} (removed and added lines are marked by the `cvs diff` command using the characters '<' and '>', respectively);
5. Accept the r_{mp} if the applied heuristics function (see below) exceeds the predefined threshold;
6. If no correlation can be found, a new *merge-point* is chosen and the steps are repeated until a possible *merge point* is found or the end of the revision tree is encountered – in either case the algorithm terminates.

Heuristics: The goal of the heuristics function is to define a measure which allows us to put a higher confidence on matching source lines which have complex structure. We have defined the following line match ratio (lmr) and complexity ratio (cmr) for assessment of *merge-point* candidates:

$$lmr = \frac{lines(S_{bp} \cap S_{mp})}{lines(S_{mp})}$$

$$cmr = \frac{complexity(S_{bp} \cap S_{mp})}{complexity(S_{mp})}$$

$lines$ is a function returning the number of lines contained in the set; $complexity()$ evaluates the "entropy" of all lines of a set (the "entropy" value for a line is determined by the length of the residuum of the transformation function specified below, where a contiguous block, i.e., a cohesive sequence of source lines, is rated higher than the same number of non consecutive matching lines).

Two thresholds have been empirically determined by running a number of tests on complex revision trees (e.g., `layout/html/style/src/nsCSSFrameConstructor.cpp` of the *Mozilla* project). Based on these two thresholds we postulate a minimum value of 0.5 for lmr and 0.3 for cmr. Since the $lines()$ function does not differentiate between matches of simple lines and complex lines, we consider lmr less trustworthy. Thus the threshold used for lmr is higher than the threshold value for cmr.

Currently we use a string compare function to identify equivalent code lines. To further improve accuracy this function may be replaced by a more sophisticated algorithm such as the approach for clone detection proposed by Casazza [9].

Transformation: Source text of C/C++ programs is transformed into a structural pattern to capture the essence of a source line which is done in the following way: (1) "background noise", i.e., short (less than 3 characters) or empty lines, are filtered and an empty string is returned; (2) white spaces are removed; (3) every word is replaced by a single character; (4) the special character ';' is removed. Figure 5 depicts the transformation of two code lines.

Example: In the following, we demonstrate the application of the algorithm on part of the revision tree of the file `layout/html/style/src/nsCSSFrameConstructor.cpp`. which is part of the *Mozilla* source tree. The algorithm itself is implemented in a script called *mergepoint*. Figure 6 depicts a snapshot of the revision tree where the search of the *merge-point* for the branch $1.638.4$ shall take place. The *branch-point* revision in our example is $r_{bp} = 1.638$. The first *merge-point* candidate ($r_{mp} = 1.641$) did not deliver any match out of the 181 possible lines and thus did not meet the threshold criteria. By taking a new date - this is done by adding two weeks to the original date - a new *merge-point* candidate ($r_{mp} = 1.654$) is selected and the values for lmr and cmr are evaluated again. In our example a full match is detected, i.e, $lmr = 1.0$ and $cmr = 1.0$ whereas the line count is 181 and the maximum complexity is 2365, thus the confidence is high that the modifications of the branch have been merged into the main trunk.

```
2001-09-28  1.637
2001-09-29  1.638
2001-10-08  |   +---1.638.4.1
2001-10-09  1.639   |
2001-10-12  |       1.638.4.2
2001-10-19  1.640   |
2001-10-20  |       1.638.4.3
2001-10-20  |       1.638.4.4
2001-10-21  1.641
.... .. ..  . ...
2001-11-07  1.654
```

Figure 6. Snapshot of revision tree

Current limitations of our approach: (1) Only inserted text is considered in the detection process. The discovery of code removal is not supported so far; (2) We assume that the merge of a branch into the main trunk happens after the branch end. Merging and afterwards undoing changes or making additional modifications to the branch which are committed to the repository, has a negative impact on the

Proceedings of the International Conference on Software Maintenance (ICSM'03)
1063-6773/03 $17.00 © 2003 IEEE

```
// Notify the parent frame}
rv = ((nsTreeRowGroupFrame*)aParentFrame)->TreeAppendFrames(newFrame);

//X
X=((X*)X)->X(X)
```

Figure 5. Transformation

detection process; (3) Changes in the source code layout, i.e., different position of line breaks in statements, have to be handled by the compare function. (4) Binary files cannot be compared with this method but are not well supported by CVS anyway; (5) Building an *Abstract Syntax Tree* (AST) and comparing the trees would be more exact but is more costly since several revisions of the source files must be checked out to build the tree (the output of `cvs diff` function is not suited for building an AST); (6) Subbranches, i.e., branches of branches, are currently not inspected.

5. Populating a release history database

Based on the data formats and structures used by CVS we designed the *Release History Database* (RHDB) that stores the extracted version and bug report data. Figure 7 depicts the database layout showing the primary entities and their relationships.

Every artefact (i.e. file) of the CVS repository has a corresponding entry in the **cvsitem** table storing the attributes extracted from the log file as described in Section 3.1. To resolve the *n:n* relationship between symbolic names (i.e. tags) and revisions of files we introduced the two entities **cvsalias** and **cvsitemalias**. Whereas **cvsalias** holds the symbolic name information, **cvsitemalias** contains a record for each entry extracted from the *symbolic names* section found in log files. Data about modification reports is stored in the **cvsitemlog** table. It contains an entry for every modification entry found in the log file. Corresponding author information is handled by **cvsauthor**.

Bug reports are directly imported from the bug tracking system into the **bugreport** table. The current attributes of this entity are derived from the *Bugzilla* system and may be extended to address further bug tracking systems. Particularly, the link of bug reports with modification reports is important for software evolution analysis. We realized this link by the **cvsitemlogbugreport** table as *n:n* relation. The table contains the bug report numbers found in the modification reports together with the respective modification report ID. Details about this import process of bug reports will be described in Section 6.

Remaining entities that are shaded in Figure 7 are not directly involved in the import process, but are used in later evolution analysis and to store the results. The entity **projstruct** reflects the hierarchical structure of the source tree. Every node hosts subtree information such as, for example, number of files or lines added/deleted in this subtree. This data is used, for instance, in the process of creating module history information for the selection of modules with a certain size. The tables **cvsitemfeature**, **feature**, and **featureset** are used in the feature evolution analysis processes which is part of our on-going and future work. The table **history** contains results which are valid for the whole evolution database such as time scale information or state information about executed queries.

To improve the communication with our RHDB and also to increase efficiency we developed a *query framework* implemented in Java. Particularly, we provide various Java classes that handle different queries according to our software evolution analysis process. The results of these queries are stored as Java objects in the tables **evalresult** and history, respectively, and can be easily accessed via the framework. In this way new data fields can be added without modifying database tables by simply specifying proper attributes in Java classes. On the contrary such a uniform access implies the use of the framework API to gain data access, hence reduces direct user-specific data manipulation capabilities via SQL queries.

6. Import process

For the import of CVS and Bugzilla data into our RHDB we use a straight forward process that basically is driven by CVS and *Bugzilla* systems but also may be adapted to other such systems. For the implementation we combined a shell script that provides the glue code, several Perl scripts for database interaction, and external commands for data retrieval. The data import process as depicted in Figure 8 consists of the following six steps:

1. The initial source code tree is created by either downloading the *Mozilla* source code packages or by checking out the source code directly from the *Mozilla* CVS repository.
2. Retrieval of log information: A shell script traverses through the source tree structure to retrieve the modi-

Proceedings of the International Conference on Software Maintenance (ICSM'03)
1063-6773/03 $17.00 © 2003 IEEE

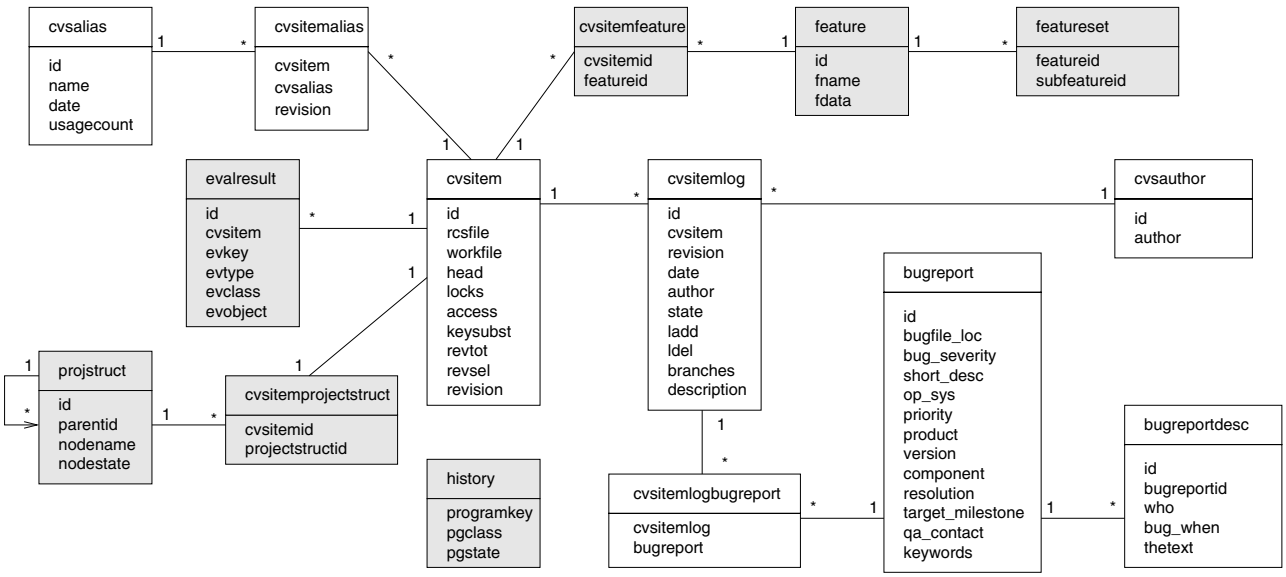

Figure 7. Release history database

fication reports from the CVS repository on directory bases (on UNIX by issuing the `cvs log -l` command). Modification reports about "unused" files that have an entry in the CVS repository but are not part of the currently checked out version are also captured (i.e., deleted files or files belonging to different products).

3. Every item of the source tree is described by the corresponding log information. The log information (revision number, modification date, modification report text, etc.) is parsed by a Perl script and stored in the evolution database, wheres the following entities of the database are populated: **cvsitem**, **cvsitemlog**, **cvsauthor**, **cvsitemalias**, **cvsalias**.

4. Bug report identifiers are extracted from the modification reports contained in the CVS log files. Since these references are not formally specified, Perl regular expression such as `bugi?d?:?=?\s*#?\s*(\d\d\d+)(.*)` or `b=(\d\d\d\d+)(.*)` are used to retrieve this information. IDs found in this step are stored in the table **cvsitemlogbugreport** so they can be used as input by the next step. Since this method does not work 100% exact, some kind of key word matching between modification report text and bug report description or a manual inspection would be required.

5. The bug report IDs are used to retrieve bug report descriptions from the *Bugzilla* database via HTTP. The external program `wget` is used to retrieve this data. All downloaded bug reports are stored on disk.

6. The XML formatted reports are parsed and the extracted bug report data is imported into the RHDB.

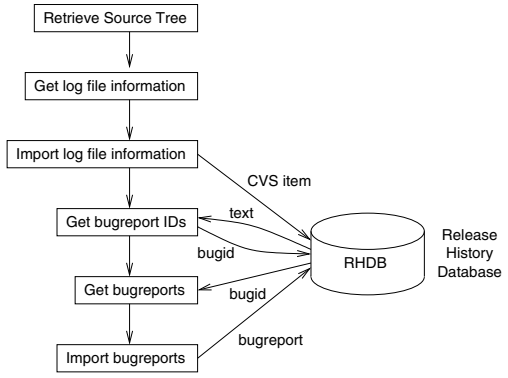

Figure 8. Import process

The result of our data import process is a populated RHDB that contains CVS version control data combined with *Bugzilla* bug report data. In terms of software evolution analysis this repository facilitates the execution of simple and complex analysis tasks as we will describe in the next section.

7. Evaluation

In this section we evaluate our approach according to import, timescale, historical, and coupling aspects of *Mozilla*.

Proceedings of the International Conference on Software Maintenance (ICSM'03)
1063-6773/03 $17.00 © 2003 IEEE

```
security/manager/pki/src/nsPKIModule.cpp
.  1  3  1  1  1  .  .  .  .  .  .  .  .  1  .  1  .  .  .  .  .  2  .  .   # of modifications
.  .  .  .  1  .  .  .  .  .  .  .  .  .  1  .  7  .  .  .  .  .  2  .  .   # of bug reports

security/manager/pki/src/nsNSSDialogs.cpp
.  2 11  9  5  .  2  1 11  1  1  3  1  .  .  1  .  .  .  .  3  3  .  1   # of modifications
.  1  1  2  4  .  2  1  9  1  2  3  1  .  .  1  .  .  .  .  2  3  .  1   # of bug reports
```

Figure 9. File change history

The results are based on data available per December 14th, 2002.

At that time 36.662 artefacts and 433.833 modification reports were imported to the **cvsitem** and **cvsitemlog** tables, respectively. From these artefacts, 23.540 were identified to have a bug report ID in one of their associated modification reports. In total 158.491 references to bug reports were found which resulted in a final number of 28.456 bug reports imported to the RHDB. Thus, out of the total number of 180.000 bug reports stored in *Bugzilla*, we filtered a solid sample of roughly a sixth of the full set. This sample exhibits an important characteristic for evolution analysis: they are referenced by modification reports and can be linked to certain changes in particular files or logically coupled files indicating what was changed, what the result of the change was, and when the change happened.

Timescale: For the analysis of evolutionary aspects, e.g., system growth or change rate, it is necessary to create a time scale based on an appropriate granularity [17]. In our semi-automatic approach we used the symbolic names retrieved from the CVS log files, e.g., `MOZILLA_1_0_RELEASE`, as indicators. During the import process each occurrence of a symbolic name is counted and the total number together with the most actual date of a modification report are stored in the RHDB (see **cvsalias**). The counting process considers symbolic names associated with the main trunk only, since they indicate the affiliation to the core architecture. These values are then used as indicators for possible release dates. A Java program selects entries by using a regular expression with groups to priorize the results, e.g., *`(MOZILLA.*RELEASE)*(.*BASE)*(.*RELEASE)*(.*)*`*, whereas only candidates with a high number of "votes" are selected. Since the *Mozilla* project team has published new releases on a nearly monthly interval, we also used a monthly interval for our further considerations.

Release history: Based on the time scale we can compute the *release history* for all artefacts in the RHDB. An artefact is marked when it first appears and the mark is updated every time the artefact is modified in one of the time slots. For example, the file *nsPKIModule.cpp* from Figure 9 has been introduced in release 33, modified in releases 34 trough 37, then again in 45 and 47, and finally in 53. This leads to the following *release sequence number*: `<33,34,35,36,37,45,47,53>`. These sequence numbers are recorded and used in the detection of logical coupling [12]. Another aspect of the release history are the number of modifications and problem reports associated with every artefact and time slot. The two example files, *nsPKIModule.cpp* (109 lines) and *nsNSSDialogs.cpp* (747 lines), in Figure 9 were introduced in release 33 (`2001-02-10`, `MOZILLA_0_8_2001020916_BASE`) and remained in the main trunk until the latest release (`2002-12-02`, `MOZILLA_1_2_1_RELEASE`). Although the first file has been modified less frequently and also has lesser problem reports, the $source - line/bugreport$ ratio is better for the second file (9.9 compared to 21), which means that the code of the first file is more error-prone. To retrieve more detailed bug report related information a simple SQL statement can be used, e.g., to list all bug reports for *nsNSSDialogs.cpp*:

```
SELECT
    b.bugreport,r.bug_severity,r.short_desc
  FROM
    cvsitem i, cvsitemlog l,
    cvsitemlogbugreport b, bugreport r
  WHERE i.id=l.cvsitem
    AND l.id=b.cvsitemlog
    AND b.bugreport=r.id
    AND i.rcsfile REGEXP 'nsNSSDialogs.cpp';
```

Besides a number of "normal" rated bug fixes (not all are shown in Figure 10), one blocking problem (blocks development and / or testing work), two critical problems (crashes, loss of data, severe memory leak), one major problem (loss of function), and two requests for enhancement were assigned to this file.

System history: From the release history of every artefact the release history of the over-all system can be derived. Figure 11 depicts 56 releases of *Mozilla* as *system history* view, which shows an approximately linear growing of the system. The rightmost bar is used for scaling and represents 100% or 34.847 artefacts. Label "0" has been assigned to the leftmost release. Due to space limitations only releases with odd numbers are labeled, and dashed lines indicate the boundary between an odd and an even labeled release. The coloring in Figure 11 indicates that about 50% of the files

Proceedings of the International Conference on Software Maintenance (ICSM'03)
1063-6773/03 $17.00 © 2003 IEEE

bugid	severity	short_desc
169943	blocker	Form submit buttons not working [embedding apps]
97044	critical	PSM is passing null string to preferences [@ nsPrefBranch::QueryObserver]
92475	critical	Need error msg for expired CRLs.
70595	major	Need to make nsIPrompt accessible to nsIChannelSecurityInfo object
44042	enhance	Wording on security-alert dialog is confusing
31896	enhance	lock icon distinguish between weak and strong encryption
169932	normal	Replace wstring with AString in IDL

Figure 10. Problem report history

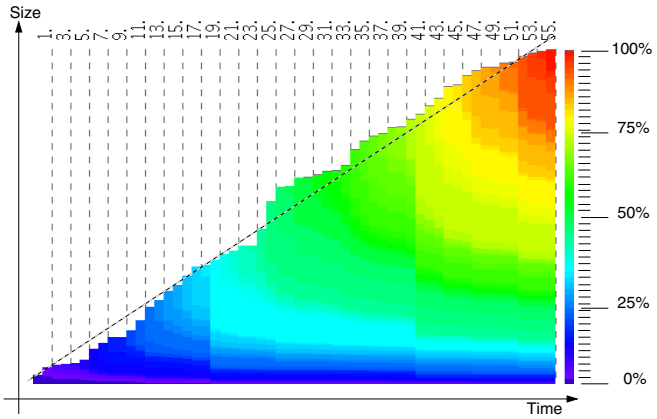

Figure 11. System history of Mozilla

have been modified within the last quarter of project duration, even though only about 25% were introduced in this last period. In [14] this approach has been applied on subsystem level to compare growing, change rate and stability. Applied on system level, it allows to compare the evolution of different systems on a very high level, e.g., to compare code maturity.

Coupling: Another aspect of a large software system is that bug reports may not be seen in isolation and thought of a problem concerning a single file. Moreover, files referenced by bug reports are logically coupled with each other either by interfaces they use, a common code base they were copied from, or features they implement. Since modifications to fix one bug reported for a specific feature often require small changes in several files, an artefact in the RHDB can be considered to be coupled with other artefacts through bug reports. The degree of coupling depends on the number of references to bug reports an artefact of the RHDB shares with other artefacts.

For instance, we selected all 33 bug reports of *nsNSSDialogs.cpp* and evaluated the number of artefacts referenced by these reports. We found 456 different artefacts which were affected by the selected bug reports. The topmost referenced files were *nsNSSCertificate.cpp* with 16 references, *nsNSSComponent.cpp* with 13 references, and *nsINSSDialogs.idl* with 11 references. Not surprising, they all belong to the same sub-module *security/manager/pki*. The first file from a different sub-module was *nsPKCS12Blob.cpp* from *security/manager/pki* with 7 references. We also found a single problem report (see [5] for report 88413) with references to 373 different files. A change in an interface of a base class required the modification of this large number of files!

These relations between bugs can be used to build groups of reports which refer to similar problems. In [11] we analyzed the effect of grouping concerning different features. The results are depicted in graphical form to support the analyst in uncovering hidden dependencies between different features.

8. Conclusions and Future Work

CVS does not provide enough mechanisms for tracking the evolution of large software systems and their particular products (reflected in branches of the CVS tree). As shown in this paper, this information can be reconstructed but a more formal mechanism supported by the version control system would be desirable. Another shortcoming of CVS is the lack of functionality to support developers with a mechanisms for linking detailed modification reports and classification of changes according to Swanson [23] or Kemerer [17]. Links to bug reports have to be added manually as free text and therefore are hard to track since authors naturally use different notions for bug report IDs. We have overcome this problem by parsing the informal information contained in modification reports and linking them with data from the bug tracking system.

Joining the modification report information with the bug report database is useful in several ways: detection of log-

Proceedings of the International Conference on Software Maintenance (ICSM'03)
1063-6773/03 $17.00 © 2003 IEEE

ically coupled files, i.e., files which are coupled by the appearance of the same bug ID in several files distributed over the source tree; identification of error prone classes with affected components or products; or estimation of code maturity with respect to the probability of remaining bugs and discovery rate of bugs in earlier releases of the system.

Data gathered from the version control system and joined with other sources will be used in our *Software Evolution Analysis* framework which is currently under development. This process will be applied on file level granularity which allows the description of evolutionary changes in terms of product line evolution and on the architectural level as well. Product line evolution can be then described in terms of feature evolution. By bridging the gap to our architecture recovery framework [16] we also will be able to reason about architecture evolution. The resulting framework enables us to answer questions such as: Which architectural design patterns / styles are used by one / some / all features (and vice versa)? What evolutionary data are available for files implementing a specific architectural style, e.g., Component Object Model? Are elements which are part of a specific architecture more error-prone than others?

A reduction of this large amount of collected historical data and visualization of this condensate is crucial for understanding of the evolutionary processes in large software projects. Using state of the art 3D real-time animation [14] or virtual reality systems will improve expressiveness of data obtained from the software evolution analysis process and will give better insight into changes on system or module level.

9. Acknowledgments

We thank the *Mozilla* developers for providing all their data for this case study to analyze the evolution of an Open Source project. Further, we thank the anonymous reviewers for their valuable comments.

References

[1] Bugzilla Bug Tracking System. `http://www.bugzilla.org/`.

[2] Microsoft Visual SourceSafe. `http://www.microsoft.com/ssafe/`.

[3] Mozilla Open-Source Web Browser. `http://www.mozilla.org/`.

[4] Rational ClearCase. `http://www.rational.com/products/clearcase/`.

[5] The Mozilla Bug Database. `http://bugzilla.mozilla.org/`.

[6] V. Ambriola, L. Bendix, and P. Ciancarini. The evolution of configuration management and version control. *Software Engineering Journal*, 5:303–310, November 1990.

[7] U. Asklund. Identifying conflicts during structural merge. In *Proceedings of NWPER'94, Nordic Workshop on Programming Environment Research, Lund, Sweden (June)*, 1994.

[8] T. Ball, J.-M. Kim, A. A. Porter, and H. P. Siy. If your version control system could talk ... In *ICSE '97 Workshop on Process Modelling and Empirical Studies of Software Engineering*, May 1997.

[9] G. Casazza, G. Antoniol, U. Villano, E. Merlo, and M. D. Penta. Identifying clones in the Linux kernel. In *Proceedings of First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 90–97. IEEE, 2001.

[10] P. Cederqvist et al. *Version Management with CVS*, 1992. `http://www.cvshome.org/docs/manual/`.

[11] M. Fischer and H. Gall. Analyzing and Relating Bug Report Data for Feature Tracking. Technical report, Distributed Systems Group, Technical University of Vienna, 2003.

[12] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance (ICSM '98)*. IEEE Computer Society Press, 1998.

[13] H. Gall, M. Jazayeri, R. R. Klösch, and G. Trausmuth. Software Evolution Observations Based on Product Release History. In *Proceedings of the International Conference on Software Maintenance 1997 (ICSM'97)*, pages 160–166, 1997.

[14] H. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: the use of color and third dimension. In *Proceedings of the International Conference on Software Maintenance (ICSM '99)*, pages 99–108. IEEE Computer Society Press, August 1999.

[15] M. Godfrey and E. H. S. Lee. Secrets from the Monster: Extracting Mozilla's Software Architecture. In *Proceedings of Second Symposium on Constructing Software Engineering Tools (CoSET'00)*, June 2000.

[16] M. Jazayeri, A. Ran, and F. van der Linden. *Software architecture for product families: principles and practice*. Addison Wesley, 2000.

[17] C. Kemerer and S. Slaughter. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25(4):493 – 509, July/August 1999.

[18] T. Kilpi. New challenges for version control and configuration management: a framework and evaluation. In *First Euromicro Conference on Software Maintenance and Reengineering*, pages 33–34, March 1997.

[19] B. Magnusson, U. Asklund, and S. Minör. Fine-Grained Revision Control for Collaborative Software Development. In *Proceedings of ACM SIGSOFT '93: Symposium on Foundations of Software Engineering*, pages 21–30, Los Angeles, California, 1993.

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

[21] T. Olsson. A View of A Merge. In *Proceedings of the Nordic Workshop on Programming Environment Research*, 1996.

[22] M. Svahnberg and J. Bosch. Evolution in Software Product Lines. *Software Maintenance*, 11(6):391–422, November–December 1999.

[23] E. B. Swanson. The dimensions of maintenance. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 492–497, 1976.

Proceedings of the International Conference on Software Maintenance (ICSM'03)
1063-6773/03 $17.00 © 2003 IEEE