Analyzing and Relating Bug Report Data for Feature Tracking

Michael Fischer, Martin Pinzger and Harald Gall
M.Fischer@infosys.tuwien.ac.at
M.Pinzger@infosys.tuwien.ac.at
H.Gall@infosys.tuwien.ac.at


Gaining higher level evolutionary information about large software systems is a key in validating past and adjusting future development processes. In this paper, we analyze the proximity of software features based on modification and problem report data that capture the system’s evolution history. Features are instrumented and tracked, the relationships of modification and problem reports to these features are established, and the tracked features are visualized to illustrate their otherwise hidden dependencies. Our approach uncovers these hidden relationships between features via problem report analysis and presents them in easy-to-evaluate visual form. Particular feature dependencies then can be selected to assess the feature evolution by zooming in into an arbitrary level of detail. Such visualization of interwoven features, therefore, can indicate locations of design erosion in the architectural evolution of a software system. Our approach has been validated using the large open source software project of Mozilla and its bug reporting system Bugzilla.

Keywords: software evolution analysis, version history, bug reports, feature analysis
Analyzing and Relating Bug Report Data for Feature Tracking

Michael Fischer, Martin Pinzger, and Harald Gall
Distributed Systems Group
Vienna University of Technology
Argentinierstrasse 8/184-1
A-1040 Vienna, Austria
{fischer,pinzger,gall}@infosys.tuwien.ac.at

Abstract

Gaining higher level evolutionary information about large software systems is a key in validating past and adjusting future development processes. In this paper, we analyze the proximity of software features based on modification and problem report data that capture the system’s evolution history. Features are instrumented and tracked, the relationships of modification and problem reports to these features are established, and the tracked features are visualized to illustrate their otherwise hidden dependencies. Our approach uncovers these hidden relationships between features via problem report analysis and presents them in easy-to-evaluate visual form. Particular feature dependencies then can be selected to assess the feature evolution by zooming in into an arbitrary level of detail. Such visualization of interwoven features, therefore, can indicate locations of design erosion in the architectural evolution of a software system. Our approach has been validated using the large open source software project of Mozilla and its bug reporting system Bugzilla.

Keywords: software evolution analysis, version history, bug reports, feature analysis.

1. Introduction

Changing requirements and technologies as the driving forces of software evolution requires the adaptation or redesign of software systems and their architectures. For large industrial or Open Source software systems data about changes is stored in versioning and bug tracking systems such as CVS [11] and Bugzilla [2]. This data provides important facts about the evolution of a software system and its architecture, hence enables engineers to learn from the past, to determine problem areas and anticipate future changes.

However, the amount of recorded data is huge and not easy to handle and therefore often not considered by engineers during maintenance. For addressing this problem we introduced the release history database (RHDB) [14] that relates versioning with bug reporting data, stores the data in a structured way, and facilitates browsing and navigation of the historical data as well as the computation of evolution metrics. The RHDB provides the basis for our further software evolution analysis tasks.

Figure 1. Grouping of Mozilla Features

In this paper, we extend the RHDB by feature data and concentrate on feature evolution analysis to identify groups and overlapping areas of feature related problem reports as depicted by Figure 1. The goal is to analyze features (such as MathML, HTTP, or About) and problem reports, relate
them across several releases, and to visualize their interrelation-ship. According to [28] we refer to a feature as an observable and relatively closed behavior or characteristic of a (software) part. Features are natural units to describe software systems from the perspective of application users, software developers, and software maintainers. Information about the implementation of features allows for reasoning about the impact of changes to features and the architectural design of software systems. By adding the dimension of evolution to this data we further can reason about future directions of feature implementations and point out problem areas that should be taken care of by software maintainers.

Our basic idea is to cluster problem report information related with a certain feature and to test the dependencies between files, which are commonly changed to fix the problem described in the problem report. The “distance” between two problem reports can be expressed as the number of files commonly modified to fix both problems. The more files they have in common the more similar the problems are.

Groups of reports can be summarized to provide a clearer picture about the problems concerning a single feature or a set of features. Moreover, hidden dependencies which are introduced through modifying groups of supposedly unrelated files from different modules are identified. Typically, such dependencies indicate bad system design or its erosion.

The contribution of this paper is a method to track features by analyzing and relating bugreport data filtered from a release history database. Features are instrumented and tracked, the relationships of modification and problem reports to these features are established, and the tracked features are visualized to illustrate their otherwise hidden dependencies. Particular feature dependencies then can be selected to assess the feature evolution by zooming into an arbitrary level of detail. Such visualization of interwoven features, therefore, can indicate locations of design erosion in the architectural evolution of a software system.

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 data source, our data model as well and validates the imported data. Following in Section 4 describe the feature extraction process and its results. The application of multidimensional scaling on release history and feature data is discussed in Section 5. We conclude in Section 6 with an indicate of future work.

2. Related Work

Mozilla has been already addressed, for example, by Mockus, Fielding and Herbsleb in a case-study about Open Source Software projects [26]. They also used data from CVS and the Bugzilla bug-tracking system but in contrast to our work focused on the overall community and development process such as code contribution, problem reporting, code ownership, and code quality including defect density in final programs, and problem resolution capacity, as well.

In [7] Bianchi et. al studied the entropy of a software system to assess its degradation by applying the entropy class of metrics on successive releases. The defined entropy class consists of several single metrics that take into account internal, external and holistic aspects of a software system. By measuring defect and maintenance effort for each metric, they are able to estimate degradation. They applied their approach to a small academic teaching project in which students had to record their activities on preprinted forms. Our approach is not based solely on the calculation of metrics and in addition we deal with real world data of a large software project.

Wong et. al [31] propose three different metrics for measurement of binding features to components or program code. They quantitatively capture the disparity between a program component and a feature, the concentration of a feature in a program component, and the dedication of a program component to a feature. Whereas they focus on the implementation of features in one single release of a software system we take into account historical data of several (all) releases to assess the evolution of features. Future work could combine both approaches.

In [23] Lanza depicts several releases of a software system in a matrix view using rectangles. Width and height of rectangles represent specific metrics (e.g. number of methods, number of instance variables of classes) according to the history of classes is visualized. Based on the evolution matrix classes are assigned to different evolution categories such as, for example, pulsar (class grows and shrinks repeatedly) or supernova (size of class suddenly explodes). Whereas he analyzes the evolution of classes we focus on features. However, a combination of both approaches could be promising.

Hsi and Potts [20] studied the evolution of user-level structures and operations of a large commercial text processing software package over three releases. Based on user interface observations they derived three primary views describing the user interface elements (morphological view), the operations a user can call (functional view), and the static relationships between objects in the problem domain (object view). As this approach does not consider a thorough code analysis, user interface issues are usually not taken into account during code analysis, a fusion with methods regarding code and release history data would yield optimum results in feature evolution analysis.

In [15, 16] our group examined the structure of a Telecommunications Switching Software (TSS) over several releases to identify logic coupling between system and sub-systems as also addressed by Bieman et. al in [8].
Based on release history data of this TSS Gall et. al presented an approach to use color and 3D to visualize the evolution history of large software systems [17]. Colors were primarily used to highlight main events of the system evolution and reveal unstable areas of the system. In the interactive 3D presentation it is possible to navigate through the structure of the system on a very coarse level and inspect several releases of the software system. Our approach extends these approaches by including feature and problem report data.

The approach described in [21] refers to Quantitative Analysis, Change Sequence Analysis, and Relation Analysis (QCR) a new Methodology for software evolution analysis. It is composed of three steps: The Quantitative Analysis is based on numerical metrics for the assessment of growth and change behavior. As second step the Change Sequence Analysis groups change events into sequences, which helps to detect common change patterns of system parts. Finally, the Relation Analysis compares classes based on change events and reveals the dependencies within the historical development of the regarded entities. The developed and adapted methods and techniques of QCR were validated based on empirical data collected from a medical software system. Findings of this work will be used to extend our Feature Evolution Analysis Model.

3. Modification and Problem Reports

Versioning information in the form of modification reports (MR), and problem reports (PR) consisting of bug data and patch information are the basic inputs for the construction of a RHDB. Typically, modification reports are retrieved from versioning systems such as CVS (Concurrent Versions System) [11] and problem reports are obtained from bug tracking systems such as Bugzilla. The release history database constitutes the basis for our further software evolution analysis activities.

<table>
<thead>
<tr>
<th>Description</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Reports imported into the RHDB</td>
<td>184625</td>
</tr>
<tr>
<td>ID of last problem report in RHDB</td>
<td>184798</td>
</tr>
<tr>
<td>Problem reports with attachments</td>
<td>49450</td>
</tr>
<tr>
<td>Attachments downloaded</td>
<td>37487</td>
</tr>
<tr>
<td>Attachments containing path information</td>
<td>26644</td>
</tr>
<tr>
<td>Bugreports with pointers to patches in attachments</td>
<td>11375</td>
</tr>
</tbody>
</table>

Table 1: Problem report data in RHDB

An important step of our data import process is the establishment of the links between MRs and PRs since CVS provides no formal mechanism for this. Such a link is constructed whenever a problem report ID number is found in an MR. These IDs are entered by authors of source code modifications as free text. Consequently, wrong links may be created because the context in which an ID is used is not clear, report IDs are incorrect or not specified at all. In all three cases the effect on the RHDB is a lower data quality.

Because these links between MRs and PRs are crucial for our analysis, we developed a link validation method that is based on the problem report ID number and the file name affected by a modification. Basically, our validation method rates the confidence of links between MRs and PRs. Problem report ID numbers in MRs are detected using a list of regular expressions. A match is rated according to the confidence value we have assigned to the expression and can be high (h), medium (m), or low (l). An expression such as “bug #42” is rated high because it definitely identifies a PR. On the contrary, a plain six digit number just appearing in the text of an MR is rated low because it also could be something else such as, for example, a date specification.

To further improve the correctness of links between MRs and PRs we check the file names specified in MRs and PRs. More precisely, we investigate patches that are attached to problem reports and check if the associated file of the modification report is referenced as target in the patch. If this is true, the rating of the problem report number is changed from ‘h’ to ‘H’, ‘m’ to ‘M’, or ‘l’ to ‘L’, respectively. All rating values are stored as the MR–PR relationship, thus can be included in queries on the release history database.

In the following we apply the link validation method on the historical data of the Mozilla project and show its results. We considered all problem reports that were entered up to version 1.3a of Mozilla (i.e. December 10, 2002). Table 1 gives an overview of problem reports and attachments imported to our release history database.

To adjust the data in the RHDB we compared the number of problem reports retrieved from Bugzilla with the number of reports referenced by modification reports as shown in Table 2. Two fields of a Bugzilla problem report are of interest for our analysis: resolution and status. A detailed description of the semantic can be found in [1].

Regarding software evolution analysis problem reports with the resolution value fixed are of major interest because fixed indicates that a solution for this problem is tested and checked into the CVS tree. A comparison of the overall number of PRs with the number of PRs referenced by MRs (CVS) indicated that a large number of PRs fall into the group of fixed reports.

Table 2 indicates that 91% of the referenced reports “ref” fall either into the group fixed/resolved (7705) or fixed/verified (18940). The other categories are sparsely filled which may indicate a possible false detection or incorrect tracking status of PRs. If we compare this data with all reports downloaded from the Bugzilla database, we recognize that a large number of PRs within the groups duplicate, invalid, won’t fix, and works for me has not been referenced. These results support our assumption in two ways: firstly, only records about PRs are made which have an effect on
the CVS repository; and second, a significant number of the identified IDs is valid if we presume that duplicate, fixed, etc. reports are equally distributed over the ordinary scale of report IDs, i.e., if IDs would be random in MRs the chances to pick a false or correct PR ID would be equal.

We now give quantitative results of the link validation method applied on the reconstructed links between MRs and PRs. In total 33499 links have been reconstructed. Our link validation method checked all these links and according the regular expression rated 27835 links as high, 4908 links as medium, and 758 links as low. Investigating the patch information 9379 of links rated high were validated that is 33%. For the other two confidence categories we obtained similar values. 3425 not validated references face 1483 validated in group medium. In group low the proportion is 496 to 260. Restricting this comparison to *c, *.cpp, and *.h files does not reveal significant differences.

Our conclusions from the above data is: references to problem reports are available in a sufficient quantity and quality to allow further analysis based on this data.

4. Feature Extraction

Goal of our feature extraction process is the gain of information about an executable program to map the abstract concept of features onto a concrete set of computational units. A computational unit can be a block, method, class, sub-module, or file. The extracted information augments the RHDB with observations about particular releases of a product and can be used to apply evolutionary analysis on the feature level. Results of the analysis greatly support the illustration of dependencies and changes in large software systems. We restricted ourselves on dynamic feature analysis, since we were basically interested on a representative set of data to validate our approach.

The used extraction process is based on Software Reconnaissance analysis technique [29, 30] and utilizes code instrumentation for its application. This approach using GNU tools [3] has been also addressed in brief in [12]. Currently, the whole process is limited to file level analysis which helps to reduce the amount of data to handle but could be extended easily onto the method level.

4.1 Scenarios and features

For feature extraction we used the scenarios defined in Table 4. Every scenario has been executed three times to obtain one set of three test-runs for every scenario. Thus a feature, e.g., HTTPS, is defined as the set of files common to all three call graphs extracted from the execution.
profiles for the same scenario. When a feature is used by several scenarios its location can be determined by using a concept lattice. Before starting extracting features from Mozilla 1.3a we created a new user profile called tuser, i.e., the profile as created by Mozilla, and changed the following options: no proxy support, no caching, blank background.

### Table 4: Scenario definitions

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Test-Runs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>0, 1, 2</td>
<td>mozilla start / blank window / stop</td>
</tr>
<tr>
<td>HTTP</td>
<td>3, 4, 5</td>
<td>TrustCenter.de via HTTP¹</td>
</tr>
<tr>
<td>HTTPS</td>
<td>6, 7, 8</td>
<td>TrustCenter.de via SSL/HTTP²</td>
</tr>
<tr>
<td>File</td>
<td>9, 10, 11</td>
<td>read TrustCenter.de from file</td>
</tr>
<tr>
<td>MathML</td>
<td>12, 13, 14</td>
<td>mathematic in Web pages³</td>
</tr>
<tr>
<td>About</td>
<td>15, 16, 17</td>
<td>“about:” protocol</td>
</tr>
</tbody>
</table>

The prepared version of Mozilla was started using the following command: `mozilla -P tuser <url>` where `-P tuser` is the name of the configuration to use and `<url>` is one of the URLs defined in the scenarios in Table 4.

### 4.2 Extraction process

Figure 2 depicts the extraction process and its intermediate outputs. Since some of the following steps are computationally intensive, it was more feasible to first create all necessary data and then to initiate the analysis process.

1. Run scenario
   - gprof.out
2. Create call graphs
   - method−name−map.txt, gprof.out
3. Parse profile information
   - binrel.txt
4. Create concept lattice
   - lattice.txt
5. Extract feature information
   - feature.dat
6. Import feature information into database

**Figure 2. Profiling Process**

1. As first step the application has to be started with one of the specified parameters to create the profile data for the scenario defined in Table 4. The result of this step is a file holding the profile data created by the GNU runtime library. Different scenario data are stored in different files and are post-processed on a file-by-file basis.

2. A modified version of the GNU gprof program [19] is used to extract the method-name-to-file-name mapping. This has to be done only once since the symbol information is static. For every scenario created in the previous step the call graph is generated using the unmodified version of GNU gprof and stored in separate files.

3. With the help of a small Perl script all call graphs are parsed and the function and method names are mapped onto file names using the mapping from the previous step. For easier manipulation and as consistency check, the file names are looked up in the RHDB and replaced by their database IDs. Outputs of this step are the binary relationships of file IDs and scenarios IDs telling which file was required by which scenario.

4. From the binary relationship data a single concept lattice [24] is generated (see Figure 4).

5. Analysis of the concept lattice identifies the computational units, i.e., in our case files, specifically required for a feature and derives a detailed relationships between features and computational units. The result of this step is a list of file IDs required to realize a specific feature.

6. Finally, a Perl script puts all file IDs, which characterizes a feature, together with with the release ID, the one under which the source files were released, into the RHDB.

### 4.3 Profiling implementation

Prerequisite for creation of profile data is the existence of an executable program with profiling support enabled. But the way to obtain a usable version was paved with pitfalls. For most of the compiling and testing we used two machines: a Pentium 4, 2GHz 512MB, RedHat 8.0 (gcc-3.2-7, libc-2.2.93) for most of the work and a Pentium II, 333MHz, 320MB, SuSE 8.1 (gcc-3.2, libc-2.2.5) & SuSE 6.2 (egcs-2.91.66, libc-2.1.1). One barrier in building older versions of Mozilla was the finding of a working compiler/library combination (see Table 5). This problem was introduced by changes in various header files. The fastest and simplest solution to solve to problem was to install a Linux distribution.

### Table 5: Mozilla & OS Version

<table>
<thead>
<tr>
<th>Product</th>
<th>Profiling</th>
<th>Build</th>
<th>Run</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mozilla 1.3a</td>
<td>gcc/glibc</td>
<td>RedHat 8.0</td>
<td>RedHat 8.0</td>
</tr>
<tr>
<td>Phoenis 0.5</td>
<td>gcc/glibc</td>
<td>RedHat 8.0</td>
<td>RedHat 8.0</td>
</tr>
<tr>
<td>Mozilla 0.9.2</td>
<td>gcc/glibc</td>
<td>SuSE 6.2</td>
<td>RedHat 8.0</td>
</tr>
<tr>
<td>Mozilla &lt; 0.9.2</td>
<td>other</td>
<td>SuSE 6.2</td>
<td>RedHat 8.0</td>
</tr>
</tbody>
</table>
shipped around the time the pertaining Mozilla package was released.

A none obvious problem was the inability of the GNU glibc to handle large amounts of profile information. A problem, it was finally fixed at the end of August 2002 (problem report 4379\(^4\)), in the GNU glibc library, originally reported by Jim Panetta\(^5\) in July 2001, causes that data are not written to disk when a program with a large number of routines is executed (a table was improperly restricted to 64K entries). This was the reason why the statically linked versions of Mozilla produced only profile data when running on RedHat 8.0 (see Table 5) but not SuSE 8.1.

Another unexpected problem was the impossibility to obtain complete call graph data when libraries were dynamically linked into the system. Prior to Mozilla release 0.9.2 static linking was not possible, only building an executable based on shared libraries was supported, which in turn causes problems with profiling since the GNU C/C++ runtime library writes the results to a fixed file location. This has two drawbacks: first, function calls which originate from outside the scope of a library cannot be traced; second, only profile data of a single library can be produced.

A solution we evaluated for early Mozilla versions was source code instrumentation using printf() statements. The necessary modifications in the source code, several thousand methods have to be instrumented, were done by an architecture recovery tool we developed for finding patterns in source code [27]. The modified version of this tool is able to find complex patterns, i.e., patterns which can’t be specified using pure regular expressions, and to replace or insert code sequences similar to UNIX’s sed or awk. Due to time limitations (human resources) we were not able to specify all patterns required for detecting all types of function and method headers. But the results were promising and we would like to further explore this method.

4.4 A feature concept lattice

Formal concept analysis is a mathematical sound technique for analyzing binary relations between a set of objects and a set of attributes [9, 18, 25]. Concept lattices are derived by concept analysis when applied on formal context. For our analysis we define the formal context as follows: computational units \(u\) are considered as the objects of the concept and scenarios \(s\) are considered as attributes; if \(u\) is executed when \(s\) is performed, then \(u\) and \(s\) are in relation, otherwise not [13]. In our study computational units are of file size granularity. For generation of the concept lattice we put the binary relationship data into a tool called concepts [24] using a customized output format. This resulting lattice data can be further processed by a graph layout program to produce a visual representation which is a directed acyclic graph.

For the specification of the required scenarios we assumed the lattice depicted in Figure 3 with a one-to-one relationship between scenarios and features except for the “HTTP” scenario. Filled circles indicate concepts which introduce new objects, open circles indicate concepts which unite a set of sub-concepts to a new concept without introducing new objects. A Comparison of this lattice with the obtained one from the Mozilla profile data (Figure 4) shall validate the assumption of the scenario data.

Before the lattice generation could start we had to remove six entries, introduced due to the indeterministic behavior of Mozilla, from the raw data file which did not appear in every of the three test-runs of a single scenario. For instance, we detected single references to nsPluginsDirUnix.cpp and nsPluginsDirUtils.h in the first test-run of scenario 4, or multiple references to nsPopupBoxObject.cpp in test-runs 2, 3, 5, 7, and 14. The resulting lattice is depicted in Figure 4. Contrary to the usual notation of using objects and attributes \(\{o_1, ..., o_j\}, \{a_m, ..., a_n\}\) for concepts, in Figures 3 and 4 we use concept id \(c_k\) and scenario \(s_i\) \(c_k; s_m, ..., s_n\) as notation.

In the following we characterize the concepts which map directly onto the defined scenarios. Concept 1 corresponds with the “HTTPS” feature from scenario 2 and adds 8 files from the sub-modules security/manager and network/socket [sic]. The “HTTP” scenario is mapped onto concept 2 which actually is a unification of three sub-concepts (”TrustCenter” display capability), 5 (JavaScript), and 8 (“Network” capability). In contrast to “HTTPS” which consists of security related files only, the expected “HTTP” information must be derived from concept 8. Concept 3 and 4 map directly onto the “File” scenario and the “About” scenario, respectively. Both concepts introduce 5 extra refer-

\(^{4}\)http://bugs.gnu.org/cgi-bin/bugzilla.pl
\(^{5}\)http://mail.mozilla.org/archive/html/mozilla-core/2001-07/msg00130.html
Figure 4. Concept lattice from profile data

In our experiments we used metric distance scaling which is a combination of Kruskal-Shepard distance scaling and metric scaling. Kruskal-Shepard distance scaling is good at achieving compromises in lower dimensions (compared to classical scaling) and metric scaling uses the actual values of the dissimilarities in contrast to non-metric scaling which considers only their ranks [10].

5.2 Data generation

PR data for the dissimilarity matrix are selected directly from the RHDB using a Java program which implements the SQL queries. Reports with more than 500 referenced files are not considered since mass-modifications are most likely concerned with administrative problems such as the correct copying license in the header of a file. These modifications are rare but have a negative impact on the optimization process. Luckily, any “major” or “critical” classified PRs are not affected by this selection.

The generation process of the dissimilarity matrix can be formally described as follows. A problem report descriptor
$d_i$ of a problem report $p_i$ is built of all artefacts $a_n$ which refer to a particular problem report via their modification reports $m_k$ (linkage MR – PR; see Section 3):

$$d_i = \{a_n | a_n \cap m_k \cap m_k \cap p_i\}.$$  

The distance data for every pair of problem report descriptor $d_i$, $d_j$ are computed according to the formula below and fed into the Dissimilarity Matrix.

$$\text{dist}(d_i, d_j) = \begin{cases} 1 & \text{if } p_i \cap p_j, \\ \frac{1}{2} \left(1 - \frac{n}{\min(s_i, s_j)}\right) & \text{if } p_i \cap p_j \end{cases}$$

where $s_i$ and $s_j$ denote the size of the descriptors $d_i$ and $d_j$ respectively. The fraction $\frac{1}{2}$ is used to emphasize the distance between unrelated objects and "weakly" linked objects. All values are scaled according to the maximum number of elements the descriptors can have in common, i.e., they are scaled to the size of the smaller one. Now we just need to define when two problem reports are linked: $p_i$ and $p_j$ in the RHDB are linked via a software artefact $a_n$ if a modification report $m_k$ exists such that

$$a_n \cap m_k \cap m_k \cap p_i$$

or two modification reports $m_k, m_l$ exist such that

$$a_n \cap m_k \cap m_k \cap p_i \cap a_n \cap m_l \cap m_l \cap p_j.$$  

5.3 Process instantiation

Since most of the default settings of Xgvis [6] were satisfying for our purpose, we had to set only the dimension of the solutions space to 2. The initial layout was manually configured by first moving all objects belonging to a certain feature into a distinct corner of the workspace. Following this manual setup phase, three optimization rounds were used to minimize the Stress function using Krsk/Sh for the first round, Classic for the second and Krsk/Sh with decreasing step-size again for the final layout. The results for the set of input data is depicted in Figure 5.

5.4 Results

For this analysis we used PRs related to the core and reports related with the features “HTTPS”, “MathML” and “About”. Due to data reduction reasons we selected altogether 2462 PRs from the RHDB for the core which were rated ‘H’ or ‘h’. This criterion has been chosen to reduce the amount of data for the optimization process since our hardware was not able to handle data sets with, e.g., 9000 data points. Other criteria, e.g., bug resolution or bug status, have not been used to reduce the amount of data. Extra large reports, i.e., as mentioned above reports with more than 500 references, were not found in the core data. PRs of the core are depicted as filled circles and constitute the background of Figure 5. The representation and number of PRs selected for the three features is as follows: the “HTTPS” feature is represented by (green) circles and consists of 170 reports; feature “MathML” is indicated by (blue) rectangles and consists of 304 reports whereas four “minor” rated reports had to be deleted due to size limit of 500 from the original set; and finally feature “About” is visualized using (pink) X and consists of 97 reports; no major or critical reports were removed but three “minor” reports. To indicate the number of files affected by a modification three different glyph sizes are used: small glyphs depict reports which are referenced; (d) ratio of code files and total files; (e) number of modified files per subdirectory; (h) number of subdirectories per module.

Figure 5. Mozilla Core & Feature Groups

To illustrate the effect of grouping we analyzed the PRs
The overlap between these two features is also depicted in Figure 1 as “feature overlapping area”. This marked area consists of 53 PRs. In total 961 C/C++ files are referenced. In contrast to the situation with “MathML” and “About”, the “HTTPS” feature is relatively independent of the two others except for a few common PRs (36756, 45797, 74803, 99163, 100476, 136756, 157136). Three PRs are common to all three features: 88413, 124042, 104158 (PRs referenced here can be checked online via the Mozilla Bug Database [5]). Also interesting is the spread of a feature or part of it over the complete core map which indicates dependencies with many different parts of the system, e.g., “About” which has no hot-spot in Figure 5.

MDS enables the visualization of dependencies between features introduced through PRs, which require modification in several files to solve a single closed problem. Degeneration in form of feature spread or interwoven features can be recognized easily.

The results of our methods were shown with a subset of features of Mozilla, but can be extended to other features rather straightforward. As a consequence, other sets of interwoven or somehow related features can be shown using the filtered bug and modification report data produced by our approach.

6. Conclusions and Future Work

The qualitative population of a release history database and augmentation with filtered problem report information is crucial for a thorough software evolution analysis. In this paper, we have shown that combining release history data with information from problem reports and their analysis offers new opportunities in the detection of otherwise hidden relationships between features. Our approach suggests first to instrument and track features, secondly to establish the relationships of modification and problem reports to these features, and thirdly to visualize the tracked features for illustrating their non apparent dependencies. Our approach uncovers these hidden relationships between features via problem report analysis and presents them in easy-to-evaluate visual form. Particular feature dependencies then can be selected to assess the feature evolution by zooming in into an arbitrary level of detail. Such visualization of interwoven features, therefore, can indicate locations of design erosion in the architectural evolution of a software system. Our approach has been validated using the large open source software project of Mozilla and its bug reporting system Bugzilla.

An interesting area for future work is the implementation of a simulation system to evaluate the impact of arbitrary design changes, for example introducing a new interface, on feature grouping. This could lead to a forecast simulating the stability of a system’s design. Another future perspective is the coupling of this visualization approach with

### Table 6: Selected areas of Figure 5

<table>
<thead>
<tr>
<th>ID</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>92</td>
<td>146</td>
<td>62</td>
<td>.42</td>
<td>25</td>
<td>7</td>
<td>5.84</td>
<td>3.37</td>
</tr>
<tr>
<td>2</td>
<td>80</td>
<td>244</td>
<td>191</td>
<td>.78</td>
<td>26</td>
<td>3</td>
<td>9.38</td>
<td>8.66</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td>52</td>
<td>46</td>
<td>.88</td>
<td>12</td>
<td>4</td>
<td>4.33</td>
<td>3.00</td>
</tr>
<tr>
<td>4</td>
<td>60</td>
<td>29</td>
<td>25</td>
<td>.86</td>
<td>15</td>
<td>5</td>
<td>1.93</td>
<td>3.00</td>
</tr>
<tr>
<td>5</td>
<td>88</td>
<td>52</td>
<td>52</td>
<td>1.00</td>
<td>2</td>
<td>1</td>
<td>26.00</td>
<td>2.00</td>
</tr>
<tr>
<td>6</td>
<td>26</td>
<td>79</td>
<td>67</td>
<td>.85</td>
<td>36</td>
<td>8</td>
<td>2.19</td>
<td>4.50</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>217</td>
<td>209</td>
<td>.96</td>
<td>117</td>
<td>23</td>
<td>1.85</td>
<td>5.08</td>
</tr>
<tr>
<td>8</td>
<td>65</td>
<td>241</td>
<td>141</td>
<td>.59</td>
<td>86</td>
<td>20</td>
<td>2.80</td>
<td>4.30</td>
</tr>
<tr>
<td>9</td>
<td>24</td>
<td>20</td>
<td>20</td>
<td>1.00</td>
<td>13</td>
<td>3</td>
<td>1.53</td>
<td>4.33</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>196</td>
<td>195</td>
<td>.99</td>
<td>111</td>
<td>25</td>
<td>1.76</td>
<td>4.44</td>
</tr>
<tr>
<td>11</td>
<td>42</td>
<td>32</td>
<td>25</td>
<td>.78</td>
<td>13</td>
<td>7</td>
<td>2.46</td>
<td>1.85</td>
</tr>
<tr>
<td>12</td>
<td>24</td>
<td>69</td>
<td>69</td>
<td>1.00</td>
<td>51</td>
<td>19</td>
<td>1.35</td>
<td>2.68</td>
</tr>
<tr>
<td>13</td>
<td>18</td>
<td>560</td>
<td>189</td>
<td>.34</td>
<td>156</td>
<td>15</td>
<td>3.58</td>
<td>10.40</td>
</tr>
<tr>
<td>14</td>
<td>37</td>
<td>189</td>
<td>174</td>
<td>.92</td>
<td>21</td>
<td>6</td>
<td>9.00</td>
<td>3.50</td>
</tr>
<tr>
<td>15</td>
<td>84</td>
<td>42</td>
<td>38</td>
<td>.90</td>
<td>7</td>
<td>2</td>
<td>6.00</td>
<td>3.50</td>
</tr>
<tr>
<td>16</td>
<td>67</td>
<td>159</td>
<td>152</td>
<td>.96</td>
<td>32</td>
<td>11</td>
<td>4.96</td>
<td>2.90</td>
</tr>
<tr>
<td>17</td>
<td>28</td>
<td>115</td>
<td>106</td>
<td>.92</td>
<td>31</td>
<td>6</td>
<td>3.70</td>
<td>5.16</td>
</tr>
<tr>
<td>18</td>
<td>141</td>
<td>85</td>
<td>79</td>
<td>.93</td>
<td>21</td>
<td>4</td>
<td>4.04</td>
<td>5.25</td>
</tr>
<tr>
<td>19</td>
<td>24</td>
<td>9</td>
<td>8</td>
<td>.89</td>
<td>5</td>
<td>2</td>
<td>1.80</td>
<td>2.50</td>
</tr>
<tr>
<td>20</td>
<td>7</td>
<td>205</td>
<td>203</td>
<td>.99</td>
<td>94</td>
<td>24</td>
<td>2.18</td>
<td>3.91</td>
</tr>
<tr>
<td>21</td>
<td>13</td>
<td>125</td>
<td>121</td>
<td>.97</td>
<td>117</td>
<td>25</td>
<td>1.06</td>
<td>4.68</td>
</tr>
<tr>
<td>22</td>
<td>13</td>
<td>312</td>
<td>312</td>
<td>1.00</td>
<td>131</td>
<td>23</td>
<td>2.38</td>
<td>5.69</td>
</tr>
<tr>
<td>23</td>
<td>28</td>
<td>53</td>
<td>49</td>
<td>.92</td>
<td>6</td>
<td>2</td>
<td>8.83</td>
<td>3.00</td>
</tr>
</tbody>
</table>

Table 7: Selected areas of Figure 1

Even though distinct features consisting of different files were selected, the analysis shows that the “MathML” and “About” share some common areas 10, 12, 13, 14, 16, 17, 22 (see Figure 5). The overlap between these two features is located in the indicated areas of Figure 5 and summarized our findings about the more interesting areas. PRs related to a single sub-module are located in (4) where we found reports about changes in the image library, e.g., PNG (Portable Network Graphics) or GIF (Graphics Interchange Format). Likewise, the areas (5), (6), and (8) have a narrow defined characteristic. They refer to PRs related to HTML parsing, network I/O, and the editor core, respectively. In area (16) we identified problems concerning printing, e.g., print preview, or crash after printing. Whereas printing is a relative small problem PRs affecting the layout and style system are located in (14), (15), (17), and (18). Indicative for interfaces changes are a low number of reports and a high number of files referenced. This happens for example in (7) where a new include file was introduced, (10), (12), (13), (20), or in (22) where part of the interface definitions of the Mozilla component model were changed.

The second feature map is depicted in Figure 1 and shows 571 PRs referenced by MRs from three features. We obtained 0.3386 as the final value for the Stress function.

<table>
<thead>
<tr>
<th>ID</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>About</td>
<td>32</td>
<td>67</td>
<td>65</td>
<td>.97</td>
<td>21</td>
<td>5</td>
<td>3.19</td>
<td>4.20</td>
</tr>
<tr>
<td>HTTPS</td>
<td>131</td>
<td>182</td>
<td>74</td>
<td>.41</td>
<td>35</td>
<td>11</td>
<td>5.20</td>
<td>3.18</td>
</tr>
<tr>
<td>MathML</td>
<td>173</td>
<td>341</td>
<td>260</td>
<td>.76</td>
<td>50</td>
<td>9</td>
<td>6.82</td>
<td>5.55</td>
</tr>
<tr>
<td>overlap</td>
<td>210</td>
<td>2648</td>
<td>2037</td>
<td>.77</td>
<td>472</td>
<td>34</td>
<td>5.61</td>
<td>13.88</td>
</tr>
</tbody>
</table>
architecture recovery systems. One possible application is to gain insight into the impact of problem reports on architectural styles and patterns. A pattern search process might identify all implementations of a socket connection. The location information is augmented with information from the RHDB and visualized using MDS.

7. Acknowledgments

We thank all the Mozilla developers for providing all their data for this case study of the evolution of an Open Source project.

References

http://www.cvshome.org/docs/manual/.