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.

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.

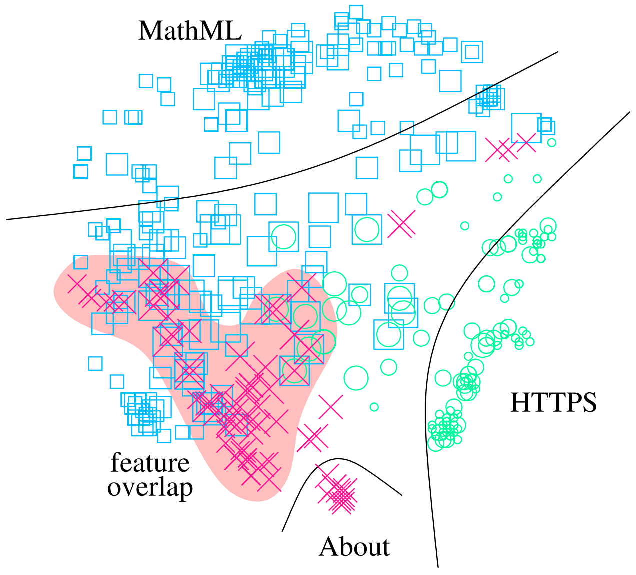

Figure 1. Grouping of Mozilla features

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 interrelationship. According to [29] 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

*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)'.

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 find out 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, as well as our data model and validates the imported data. In Section 4 we 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 indication 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 [27]. 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 [32] 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 [24] Lanza depicts several releases of a software system in a matrix view using rectangles. Width and height of rectangles representing specific metrics (e.g. number of methods, number of instance variables of classes) according to the history of classes are 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 [21] 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 good 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 [18, 22] refers to *Quantitative Analysis, Change Sequence Analysis, and Relation Analysis* (QCR) as 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

The 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 1. Problem report data in RHDB

Description	#
Problem Reports imported into the RHDB	184625
ID of last problem report in RHDB	184798
Problem reports with attachments	49450
Attachments downloaded	37487
Attachments containing patch information	26644
Bugreports with pointers to patches in attachments	11375

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 might be 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 as *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 semantics 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 positive 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

Table 2. Problem reports of Bugzilla

\ Status Resolution	undefined all / ref	new all / ref	closed all / ref	assigned all / ref	reopend all / ref	resolved all / ref	verified all / ref	unconfirmed all / ref
undefined	1318 / 199	19582 / 415	0 / 0	7016 / 510	988 / 173	0 / 0	0 / 0	3348 / 16
duplicate	0 / 0	0 / 0	282 / 5	0 / 0	0 / 0	13855 / 66	41648 / 487	0 / 0
fixed	0 / 0	0 / 0	333 / 63	0 / 0	0 / 0	14806 / **7705**	36620 / **18940**	0 / 0
invalid	0 / 0	0 / 0	315 / 6	0 / 0	0 / 0	4551 / 43	9116 / 113	0 / 0
later	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	4 / 4	6 / 5	0 / 0
moved	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	24 / 0	73 / 1	0 / 0
remind	0 / 0	0 / 0	3 / 0	0 / 0	0 / 0	1 / 1	5 / 5	0 / 0
won't fix	0 / 0	0 / 0	92 / 1	0 / 0	0 / 0	1823 / 46	3123 / 69	0 / 0
works for me	0 / 0	0 / 0	359 / 2	0 / 0	0 / 0	9765 / 93	15569 / 308	0 / 0

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.

Table 3. Products and problem reports

	fixed reports			all reports		
Product	all	found	%	all	found	%
Browser	35520	20396	57.42	129889	22208	17.10
Bugzilla	1630	9	0.55	4564	26	0.57
Calendar	371	192	51.75	709	197	27.79
Chimera	87	86	98.85	96	95	98.96
Derivatives	1	0	0.00	23	1	4.35
Directory	189	93	49.21	380	103	27.11
Documentation	225	8	3.56	522	10	1.92
MailNews	7089	4583	64.65	29112	4978	17.10
Phoenix	208	8	3.85	1180	9	0.76
Tech Evangelism	1314	6	0.46	4908	16	0.33
Webtools	239	1	0.42	617	2	0.32
mozilla.org	1126	5	0.44	2122	15	0.71

A summary of PRs in the RHDB with respect to different product categories defined in Bugzilla for the *Mozilla Application Suite* [4] is given in Table 3. Results for reports having status *fixed* are listed in the left column, whilst the right column labeled "all reports" lists the number of reports in the database regardless of their status. In the most important category "Browser" we found 20396 of 35520 downloaded reports which yields to a success rate of 57.42%. Interesting are also other categories such as Tech Evangelism, Webtools, or mozilla.org which may be used as indication for false positive detections which is less than 1%.

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

The 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 [30, 31] 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

Scenario	Test-Runs	Description
Core	0, 1, 2	mozilla start / blank window / stop
HTTP	3, 4, 5	TrustCenter.de via HTTP[1]
HTTPS	6, 7, 8	TrusterCenter.de via SSL/HTTP[2]
File	9,10,11	read TrustCenter.de from file
MathML	12,13,14	mathematic in Web pages[3]
About	15,16,17	"about:" protocol

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.

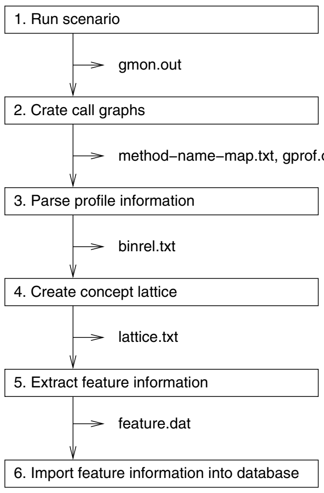
1. Run scenario
gmon.out
2. Crate call graphs
method−name−map.txt, gprof.out
3. Parse profile information
binrel.txt
4. Create concept lattice
lattice.txt
5. Extract feature information
feature.dat
6. Import feature information into database

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 [20] 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 [25] 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 the 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 this problem was to install a *Linux* distribution shipped around the time the pertaining *Mozilla* package was released.

Table 5. Mozilla and OS version

Product	Profiling	Build	Run
Mozilla 1.3a	gcc/glibc	RedHat 8.0	RedHat 8.0
Phoenix 0.5	gcc/glibc	RedHat 8.0	RedHat 8.0
Mozilla 0.9.2	gcc/glibc	SuSE 6.2	RedHat 8.0
Mozilla $<$ 0.9.2	other	SuSE 6.2	RedHat 8.0

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

[1] http://www.trustcenter.de/
[2] https://www.trustcenter.de/
[3] http://www.w3.org/Math/testsuite/testsuite/General/Math/math3.xml
[4] http://bugs.gnu.org/cgi-bin/gnatsweb.pl

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 can not 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 [28]. The modified version of this tool is able to find complex patterns, i.e., patterns which cannot 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, 19, 26]. 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* [25] 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

[5]http://mail.gnu.org/archive/html/bug-glibc/2001-07/msg00130.html

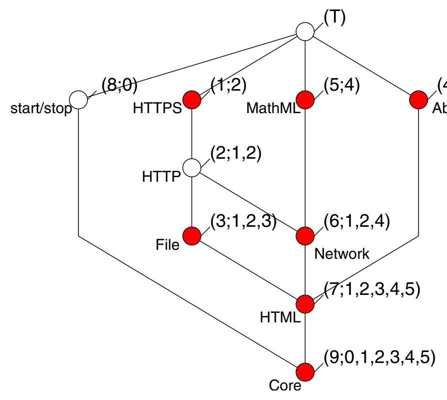

Figure 3. Assumed concept lattice

unite a set of 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.

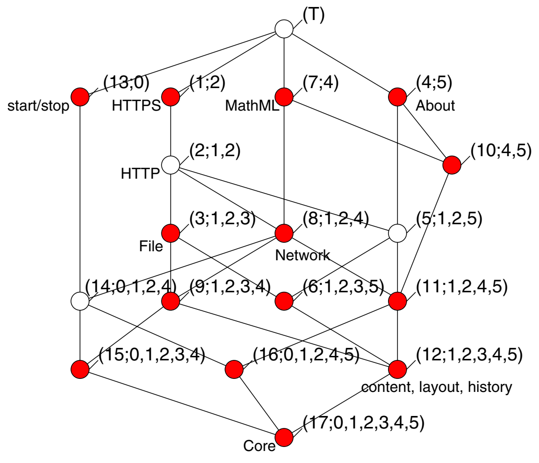

Figure 4. Concept lattice from profile data

Contrary to the usual notation of using objects and attributes $(\{o_i, ..., o_j\}, \{a_m, ..., a_n\})$ for concepts, in Figure 3 and Figure 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 *netwerk/socket* [sic!]. The "HTTP" scenario is mapped onto concept **2** which actually is a unification of three sub-concepts **3** ("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 references. Besides other features, compression (zlib), Portable Network Graphics (PNG), required by the "MathML" test case, concept **7** adds XSL Transformations (XSLT) from the sub-module *extensions/transformiix* [sic!] to the concept lattice. XSLT is a language for transforming XML (Extensible Markup Language) documents into other XML documents.

The "Core" scenario introduces concept **13** which is used by start/stop only and adds 2 files: *nsPlaintextDataTransfer.cpp* and *nsTimeBomb.cpp*. This is surprising since we expected no extra files to be added by this concept. The most specialized concept **17** consists of all objects which are common to all scenarios. For *Mozilla 1.3a* and our setting these are 705 files from 136 directories divided into 26 modules.

From the remaining concepts we characterize the more interesting ones in brief: on top of the lattice is the largest concept located. It is indicated by ($\top$) and represents a unification of all sub-concepts below. It comprises 959 references to objects, i.e., source files. Concept **8** primarily adds 18 network and protocol related files which corresponds with the assumption of a concept providing this service to the "HTTP" and "MathML" scenario. The largest sub-concept, except for the Core, is concept **12** which adds 99 referenced files from 7 different modules. Basically content, layout, and browsing history related files are added. This corresponds with our assumption of a "HTML" feature in Figure 3.

We conclude that the obtained lattice validates the assumption and that the feature data gained can be accepted for the next processing step.

4.5. Results

Extraction of call graph data using GNU tools was unexpectedly difficult in terms of process application and resource consumption for a software package of this size. Call graph data from first *Mozilla* version are still not available but would be required to build an accurate picture of earlier program versions.

With a mixture of compiler supported profiling and manual instrumentation of source code, it is possible to gain the required runtime information to enable feature extraction from current releases and historical source code.

5. Grouping and relating features

From the data in the RHDB we selected the "Core" and three features ("HTTPS", "MathML" and "About") for finding groupings in PR data using multidimensional scaling (MDS) [23].

5.1. Overview MDS

The goal of MDS is to map objects $i = 1, \ldots, N$ to points $\|\mathbf{x}_i - \mathbf{x}_j\| \in \mathbb{R}^k$ in such a way that the given dissimilarities $D_{i,j}$ are well-approximated by the distances $\|\mathbf{x}_i - \mathbf{x}_j\|$ whereas k is the dimension of the solution space. MDS is defined in terms of minimization of a cost function called *Stress*, which is simply a measure of lack of fit between dissimilarities $D_{i,j}$ and distances $\|\mathbf{x}_i - \mathbf{x}_j\|$. In its simplest case, *Stress* is a residual sum of squares:

$$\text{Stress}_D(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \left(\sum_{i \neq j} (D_{i,j} - \|\mathbf{x}_i - \mathbf{x}_j\|)^2 \right)^{\frac{1}{2}}$$

where the outer square root is just a convenience that gives greater spread to small values [10].

For 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):

$$d_i = \{a_n | a_n \mathsf{R} m_k \wedge m_k \mathsf{R} p_i\}.$$

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*.

$$\text{dist}(d_i, d_j) = \begin{cases} 1 & \text{if } p_i \not\mathsf{R} p_j, \\ \frac{1}{2}(1 - \frac{n}{\min(s_i, s_j)}) & \text{if } p_i \mathsf{R} p_j \end{cases}$$

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

$$a_n \mathsf{R} m_k \wedge m_k \mathsf{R} p_i \wedge m_k \mathsf{R} p_j$$

or two modification reports m_k, m_l exist such that

$$a_n \mathsf{R} m_k \wedge m_k \mathsf{R} p_i \wedge a_n \mathsf{R} m_l \wedge m_l \mathsf{R} p_j.$$

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 solution 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 < 10 times; medium sized glyphs symbolize that the number of references is in the range $[10..100]$; large glyphs represent reports which are referenced > 100 times.

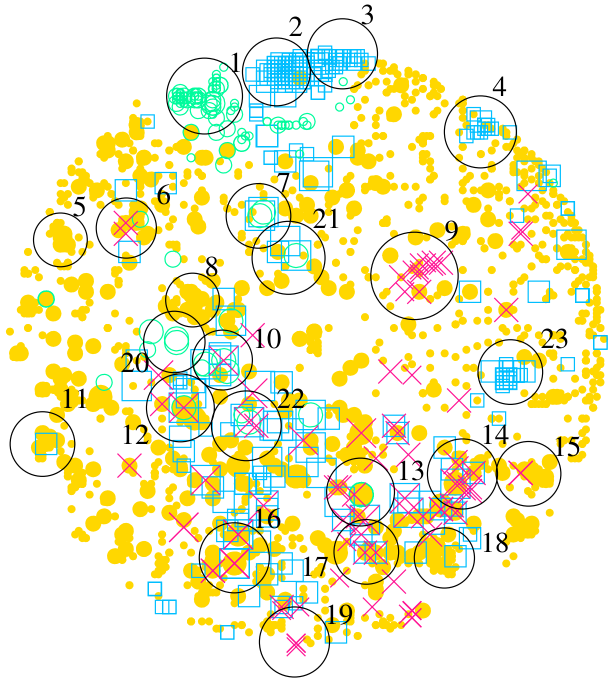

Figure 5. Mozilla core & feature groups

The result of the optimization process, final value of the *Stress* function was 0.4056, is depicted in Figure 5. Individual results of the indicated areas - the selection of the data objects had to be done manually - are listed in Table 6 whereas the columns have the following meaning: (**a**) number of problem reports in the indicated area; (**b**) number of total files referenced; (**c**) number of *.h, *.c, *.cpp files referenced; (**d**) ratio of code files and total files; (**e**) number of different subdirectories; (**f**) number of different modules, i.e., first level directories; (**g**) number of modified files per subdirectory; (**h**) number of subdirectories per module.

To illustrate the effect of grouping we analyzed the PRs located in the indicated areas of Figure 5 and summarized our findings about the especially 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

Table 6. Selected areas of Figure 5

ID	a	b	c	d	e	f	g	h
1	92	146	62	.42	25	7	5.84	3.57
2	80	244	191	.78	26	3	9.38	8.66
3	36	52	46	.88	12	4	4.33	3.00
4	60	29	25	.86	15	5	1.93	3.00
5	88	52	52	1.00	2	1	26.00	2.00
6	26	79	67	.85	36	8	2.19	4.50
7	11	217	209	.96	117	23	1.85	5.08
8	65	241	141	.59	86	20	2.80	4.30
9	24	20	20	1.00	13	3	1.53	4.33
10	7	196	195	.99	111	25	1.76	4.44
11	42	32	25	.78	13	7	2.46	1.85
12	24	69	69	1.00	51	19	1.35	2.68
13	18	560	189	.34	156	15	3.58	10.40
14	37	189	174	.92	21	6	9.00	3.50
15	84	42	38	.90	7	2	6.00	3.50
16	67	159	152	.96	32	11	4.96	2.90
17	28	115	106	.92	31	6	3.70	5.16
18	141	85	79	.93	21	4	4.04	5.25
19	24	9	8	.89	5	2	1.80	2.50
20	7	205	203	.99	94	24	2.18	3.91
21	13	125	121	.97	117	25	1.06	4.68
22	13	312	312	1.00	131	23	2.38	5.69
23	28	53	49	.92	6	2	8.83	3.00

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 interface 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 7. Selected areas of Figure 1

ID	a	b	c	d	e	f	g	h
About	32	67	65	.97	21	5	3.19	4.20
HTTPS	131	182	74	.41	35	11	5.20	3.18
MathML	173	341	260	.76	50	9	6.82	5.55
overlap	210	2648	2037	.77	472	34	5.61	13.88

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 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 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 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] A Bug's Life Cycle. http://bugzilla.mozilla.org/bug_status.html.

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

[3] GNU's Not Unix! - the GNU Project and the Free Software Foundation (FSF). http://www.gnu.org/.

[4] Mozilla Branding. http://www.mozilla.org/roadmap/branding.html.

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

[6] XGobi: A System for Multivariate Data Visualization. http://www.research.att.com/areas/stat/xgobi.

[7] A. Bianchi, D. Caivano, F. Lanubile, and G. Visaggio. Evaluating Software Degradation through Entropy. In *7th International Software Metrics Symposium*, November 2001.

[8] J. Bieman, A. Andrews, and H. Yang. Understanding change-proneness in OO software through visualization. In *Proceedings of 11th International Workshop on Program Comprehension*. IEEE, 2003.

[9] G. D. Birkhoff. *Lattice Theory*. American Mathematical Society, 1967.

[10] A. Buja, D. F. Swayne, M. Littman, N. Dean, and H. Hofmann. XGvis: Interactive Data Visualization with Multidimensional Scaling. *Tentatively accepted for publication in the Journal of Computational and Graphical Statistics*, 2001. http://www.research.att.com/areas/stat/xgobi/papers/xgvis.pdf.

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

[12] T. Eisenbarth, R. Koschke, and D. Simon. Aiding Program Comprehension by Static and Dynamic Feature Analysis. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society Press, November 2001.

[13] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *Transactions on Software Engineering*, 29(3):210–224, 2003.

[14] M. Fischer, M. Pinzger, and H. Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proceedings of the 2003 International Conference on Software Maintenance (ICSM 2003), Amsterdam, Netherlands*, September 2003.

[15] 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.

[16] 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.

[17] 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.

[18] H. Gall, J. Krajewski, and M. Jazayeri. Cvs release history data for detecting logical couplings. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE), Helsinki, Finland*. IEEE Computer Society Press, September 2003.

[19] B. Ganter and R. Wille. *Formal Concept Analysis*. Springer, 1999.

[20] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a Call Graph Execution Profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[21] I. Hsi and C. Potts. Studying the Evolution and Enhancement of Software Features. In *Proceedings of the 2000 IEEE International Conference on Software Maintenance*, pages 143–151, 2000.

[22] J. Krajewski. QCR - A methodology for Software Evolution Analysis. Master's thesis, Technical University of Vienna, June 2003.

[23] J. B. Kruskal and M. Wish. Multidimensional Scaling. *Quantitative Applications in the Social Sciences*, 11, 1978.

[24] M. Lanza. The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE)*, 2001.

[25] C. Lindig. Compute Concept Lattice from Relation, 1998. http://www.eecs.harvard.edu/~lindig/src/concepts.html.

[26] C. Lindig and G. Snelting. *Begriffliche Wissensverarbeitung. Methoden und Anwendungen*, chapter Formale Begriffsanalyse im Software Engineering. Springer, 1999.

[27] 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.

[28] M. Pinzger, M. Fischer, H. Gall, and M. Jazayeri. Revealer: A Lexical Pattern Matcher for Architecture Recovery. In *9th Working Conference on Reverse Engineering (WCRE)*, October 2002.

[29] E. Pulvermüller, A. Speck, J. O. Coplien, M. D'Hondt, and W. DeMeuter. Feature Interaction in Composed Systems. In *Feature Interaction in Composed System*, 2001.

[30] N. Wilde, J. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *International Conference on Software Maintenance*, pages 200–205, 1992.

[31] N. Wilde and M. C. Scully. Software Reconnaissance: Mapping Program Features to Code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, January 1995.

[32] W. E. Wong, S. S. Gokhale, and J. R. Horgan. Quantifying the closeness between program components and features. *The Journal of Systems and Software*, 54(2):87–98, 2000.