

Integrating Version Data with Feature and Source Model Data for Architecture Evolution Analysis *

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

Abstract

Large and complex software systems are costly and are designed to have a long life expectancy. Design for change and keeping the architecture sound during maintenance phases is a key for cost reduction. Therefore, analyzing how software systems evolved in the past is beneficial and completes the picture of the current architecture obtained by architecture recovery techniques. In this paper we propose an architecture evolution analysis approach that concentrates on the investigation of the architecture in terms of architectural primitives, their implementation by source elements and their evolution. By combining the spaces of architecture recovery and version analysis our approach produces multidimensional views on software architectures that indicate architectural shortcoming and points of design erosion. We demonstrate our approach by applying it to the large Mozilla open source software project and report on the results.

Keywords: software evolution analysis, software architecture, architecture recovery, version history, bug reports

1. Introduction and motivation

Large complex software systems undergo a lot of changes during their life-cycle that are carried out as maintenance or evolution tasks. According to the studies of [4] these changes make up to 70% of the total costs of software projects. Hence, keeping these costs low is a key objective of companies.

Concerning the reduction of costs a good architectural

design, its documentation and the conformance of implementation to the architecture are key prerequisites. Currently, reverse engineering methods such as architecture recovery are used to investigate software systems with respect to its as-is architecture.

However, architecture recovery methods so far mostly consider source code structures to identify a particular software architecture, but do not take into account any evolutionary data to discover logical couplings between architectural elements. In our previous works we have shown that version information such as bug reports or modification reports contain valuable development information that, when analyzed, indicates architectural shortcomings or points of design erosion [13, 12]. We address very large systems such as Mozilla in which pure source code analysis methods usually fail due to the size of the investigated system.

Our approach filters architectural primitives in source models, delivers their instances and allows the engineer to concentrate on these instances for further analyses. Evolution data of these instances (on a file level) is combined to investigate logical couplings with "surrounding" source model structures (e.g. classes, functions, files, etc.). Together, these two information spaces complete the picture of an architectural primitive. We, therefore, propose to analyze a software architecture in a multi-dimensional way: from source models and along the version history of the system.

Regarding the version data we retrieve modification reports (MRs) from versioning systems such as CVS [28] and problem reports (PRs) from bug tracking systems such as Bugzilla [38]. Our populator tool computes links between both reports and stores them to the release history database (RHDB). Architectural primitives are extracted from source models applying our architecture recovery approach. In addition, our approach takes into account features that are extracted from execution traces. Both, data about architec-

*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)'.

tural primitives and features are stored to the database and together with the RHDB database form the different information spaces that are input to our architecture analysis approach.

The remainder of this paper is organized as follows: Section 2 introduces our architecture evolution analysis process. In Section 3 we describe the preparation of the evolution databases. Section 4 is concerned with a detailed description of the analysis techniques used by our process. In Section 5 we demonstrate our approach by applying it to the Mozilla case study. Related work is provided in Section 6 and Sections 7 draws the conclusions and indicates future work.

2. Information spaces in architecture evolution analysis

The basic idea of our architecture evolution analysis approach is to relate the information extracted by our architecture recovery approach with information from the software evolution analysis process. Architecture recovery reveals data about architectural primitives under study and maps them to source code. Basically, these primitives are concerned with certain architectural aspects or features of the software system. In terms of Mozilla, examples for such aspects are communication, security or content handling and presentation. Retrospective analysis on the other hand addresses the version and bug history of software systems and reveals logical couplings between source code organization entities such as source sub-systems, modules, and files.

To bridge the gap between the architecture recovery space and retrospective analysis the selection of an entity that is common to both spaces and has equal semantic is mandatory. Because historical data basically is stored on the level of files we use source files as such entities. In terms of architecture primitives they contain the source model elements implementing the primitives. Concerning the historical data files are associated with version and bug reporting information. Therefore, source files and source directories respectively are predestinated for linking both spaces. Source model elements such as classes or methods are also possible but need reasonable work around to link them to the release history data. In this work we concentrate on source files as linking elements.

For the integration of architecture including feature analysis and retrospective analysis results into a common analysis process we introduce our iterative and interactive architecture evolution analysis process as depicted by Figure 1.

Basically, our architecture evolution analysis process consist of two major phases: database generation and architecture evolution analysis. In the following sections we describe each process phase.

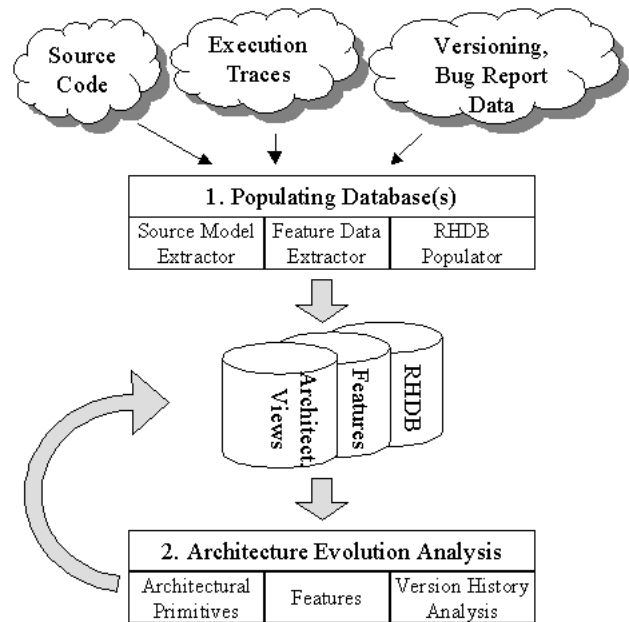


Figure 1. Architecture Evolution Analysis Process (Overview).

3. Evolution data preparation

The first process phase is concerned with the preparation of the fact database(s) comprising source model, feature, versioning and bug reporting data. They provide the basic input for the architecture evolution analysis phase.

Source Model Extractor: For the extraction of source models from source code and related files (e.g. configuration files, meta data files) we use static analysis techniques such as parsing and lexical-based tools. The latter address information not considered by parsers for completing the source models. A numerous amount of commercial and research tools are available such as, for example, Imagix-4D [18], SourceNavigator [33], or cppx [7].

Typically, each tool uses its own database and format for storing the extracted data that can not be accessed/processed by other tools. Hence, defining a data format that is common to other tools is mandatory. Our approach is based on the FAMIX [32] meta-model and extracts data in Rigi Standard Format (RSF) [34]. RSF data is easy to process, most of the analysis and visualization tools are capable of it, and there are a various converters available to convert data to and from RSF files.

Feature Extractor: Goal of the feature extraction process is to gain the necessary information to map the abstract concept of features onto a concrete set of files which implement a certain feature. To extract the required feature

data we apply the software reconnaissance technique as described in [35, 36]. Feature data is extracted by processing feature specific execution traces generated by profiling tools such as, for example, gprof. A trace is a call-graph containing all functions invoked and the source files implementing the recorded functions. For each feature under study an execution trace is generated and the source model information is extracted and stored in the feature database.

Although, the feature extraction process is straight forward configuring the compiler, linker and the profiler can be complicated. We refer the reader to our previous work [12] that contains a more detailed discussion of feature extraction.

RHDB Populator: Release history data is retrieved from versioning systems such as the concurrent versioning system (CVS) [28] and bug tracking systems such as Bugzilla [38]. In particular, we obtain modification reports (MRs) from CVS and problem reports (PRs) from Bugzilla. Figure 2 depicts the nucleus of our *Release History Database* with the coupling of modification reports and problem reports that are stored to the *cvsitemlogbugreport* table. MRs are stored in the *cvsitemlog* entity and PRs in the *bugreport* entity of our RHDB. Information about the file to whom a MR belongs is stored in *cvsitem*.

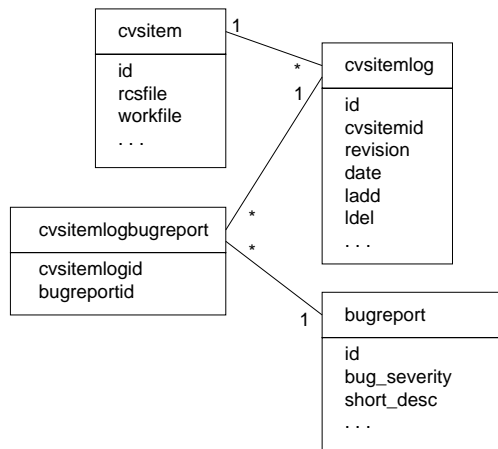


Figure 2. “Nucleus” of the RHDB

Establishing the links between MRs and PRs is an important issue of the RHDB population process. Depending on the versioning system these links are either provided by the system or have to be established separately. Concerning CVS and Bugzilla, this needs to be done separately. A link is stored whenever a reference to a problem report is found in a modification report. Bug report numbers in modification reports are searched by regular expressions, e.g., bug #42. Because these numbers are entered as free text results contain correct and false positive matches as well. To improve data quality all matched numbers are validated using

information available with PRs and secondary information such as patches [12].

4. Architecture evolution analysis

All three information sources stand-alone give insight into basic architectural properties of software systems such as architecture primitives and features implemented, or files and directories frequently changed and/or tightly logical coupled. In this manner each information space can be addressed separately or in various combinations with the other spaces. In any case the objective is to abstract different views each showing particular aspects of the software architecture that are mandatory to understand, assess, maintain, and evolve the system. In the following subsections we describe the techniques we use in our process for architecture evolution analysis.

4.1. Recovery of architectural primitives

According to the work of Mehta and Medvidovic [24] a software architecture can be recursively decomposed into architectural primitives that are *fine-grained, low-level abstractions, each of which focuses on a single aspect of architectural styles identified above*. From the perspective of architecture recovery these primitives are first-class entities to be reconstructed.

In our architecture recovery approach we concentrate on the extraction of architectural primitives from various artifacts available for a software system. Artifacts include source code, configuration files, design documentation, expert knowledge, and the running system. Because of the mass of information and its complexity the recovery process is iterative and controlled by the user. The role of the user is primarily to check resulting views and drive the recovery process towards views relevant for the analysis task at hand.

The architectural primitive recovery process bases on the source model data extracted during the database generation phase and consists of four steps:

1. Pattern identification
Identify “patterns” that are used to realize the architectural primitive (e.g. source code patterns). They are hot-spots and provide the starting points for subsequent recovery steps.
2. Pattern matching
Match pattern specification with source models and determine instances of hot-spots.
3. Information abstraction
Investigate relationships of identified hot-spots and determine associated source model elements that can be aggregated to more abstract elements.

4. View visualization

Represent extracted and abstracted elements to allow for user interaction and control.

For the identification of hot-spots indicating an architectural primitive design documents or experts are a fundamental source of information. Basically, such hot-spots are specific identifiers, source code statements or structures, or configuration parameters used in the implementation of the system. Identifiers offer important semantic information such as, for example, an affiliation to a component, especially, when certain naming conventions have been used for the implementation (e.g., names of directories, files, classes and methods). For example, according to the documentation of Mozilla the function identifiers `PR_NewTCPSocket` and `SSL_ImportFD` represent hot-spots of the secure communication primitive.

In addition, design patterns as described by Gamma et al. [15] and Schmidt et al. [31] represent considerable design decisions that are subject to recover.

To locate the hot-spots in source files two techniques are applicable:

- Pattern matching.
- Source code browsing.

To match specific identifiers, code statements, configuration parameters, and basic source code structures regular expressions tools such as *grep* or query facilities shipping with reverse engineering tools yield good results. However, more complex source code structures or design patterns need the application of pattern matching tools.

Regarding source code patterns we apply our extended string pattern matching tool *mpgrep* that is an extended version of the *Revealer* tool [29]. It facilitates the definition of simple and complex pattern definitions by composing several pattern elements in a regular expression like way. Pattern elements are specified in terms of functions that may take several parameter values. Each element is assigned a unique identifier to “call” pattern elements within other pattern element definitions. The root element is the entry point of the pattern definition and called by the pattern processing routine. Pattern definitions are stored to files that form the pattern repository. The matching process detects pattern instances and outputs corresponding data in RSF format.

The extraction of design patterns from source code is a common research problem addressed by several approaches such as, for example, Grok [9], Pat [21], SPOOL [20], FUJABA [27]. Basically, pattern definitions used by these approaches specify the structure of design patterns in terms of the elements (i.e. classes, methods and attributes) and relationships between the elements (i.e. inheritance, association, method calls) constituting the pattern. Based on the pattern definition the matching process queries the source

model for instances. Because of the variation of design pattern implementations the result also contains a considerable number of false positives and negatives that have to be sorted out by the user. The pattern matching step results in a list of hot-spots expressed in terms of source model entities and source code location data. They are the basic source elements involved in the implementation of an architectural primitive.

To complete the picture of an architectural primitive the surrounding of matched hot-spots is investigated along their relationships with other source model entities. Basically, all different relationships are of interest because they imply dependencies between entities. However, to maintain a reasonable set of entities the user, depending on the analysis task and the architectural primitive to recover, selects a subset of relationships to be considered. For example, in the Mozilla case study we followed the call-graph of the `PR_NewTCPConnection` function to determine the entities realizing the communication primitive.

The result is a set of source model entities that realize an architectural primitive. Often the amount of entities is enormous hence aggregation to a certain level is mandatory. The selection of this level depends on the degree of detail needed and ranges from methods, classes up to the level of files and directories. For the analysis of big systems such as Mozilla an aggregation of the information up to the level of directories is reasonable. Typically, they can be directly mapped to components or subsystems of a software system. In our architecture recovery process we use the Grok [9] tool and a number of scripts to aggregate and group information automatically. Additionally, we use the facilities of Imagix-4D and Rigi for manual aggregation.

The outcome of the architectural primitive extraction is a list of architectural primitives and relevant source model entities comprising the primitive, and relationships between them. The information is used to generate views on the system’s software architecture that basically show structural aspects – which primitive is used where and implemented how. Such views aid in analyzing and assessing the software architecture of one particular release *as-is*. But, it lacks of important historical data that brings into the notion of evolution.

4.2. Feature analysis

The feature analysis process is concerned with relating features to version information and visualizing the results. Because of the huge amount of information adequate techniques are mandatory that group and cluster information. Therefore, we concentrate on these techniques including concept analysis [3, 16] and multi-dimensional scaling (MDS) [22].

Formal concept analysis is a mathematical sound tech-

nique for analyzing binary relations between a set of objects and a set of attributes. Concept lattices are derived by concept analysis when applied on formal context. For our analysis we define the formal context as follows: computational units μ are considered as the objects of the concept and scenarios σ are considered as attributes; if μ is executed when σ is performed, then μ and σ are in relation, otherwise not [8]. 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 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.

The second technique we use for analyzing and visualizing the data is multidimensional scaling (MDS) which is implemented in a software package called *xgvis*. The goal of MDS is to map objects $i = 1, \dots, N$ 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 $\|\mathbf{x}_i - \mathbf{x}_j\|$ [5]. Input to this processing step are the node data and the dependencies between files as weighted edges, whereas edges introduced by the directory structure were rated 10 and a common problem report was rated 1.

The result of latter approach is a graph in which nodes with high coupling are placed together. For more details on feature evolution analysis we refer the reader to our previous work [12].

4.3. Integration of information spaces

Both techniques described in previous sections yield data about architectural primitives and features implemented. For adding the facts about evolution these results have to be integrated into a common result set. As described previously we base on source files for combining all three information spaces because they are inherent in the representation of architectural primitives, features, and version information.

Source model information about recovered architectural primitives is available in RSF format. Features are represented in terms of source files that contain the functions executed and stored in the release history database (RHDB). Version information also is available from the RHDB and can be retrieved from there by SQL queries.

Concerning version information we query the database with respect to the dependencies due to bug report data. The fundamental principle of our approach is:

Two source files are logically coupled if there is at least one problem report (PR) that contains references to both files.

A reference to a file corresponds to a reference to a `cvsitem` that identifies each source file stored in the CVS repository respectively. The result of the RHDB queries is retrieved in the format:

```
<cvsitemid1><cvsitemid2><count><bugreportids>
<rscfile1><rscfile2>
```

`cvsitemid1` and `cvsitemid2` contain the identifiers of the CVS items coupled. The strength of coupling, that is the number of PRs referencing both files, is stored in `count`. `bugreportids` contains the list of problem reports affected. Latter two items `rscfile1` and `rscfile2` are obtained from the `cvsitem` table and contain the full name of the source files (see Figure 2).

Two features are logically coupled if there is a logical coupling between the source files implementing the two features.

For the determination of logical coupling of features the RHDB is queried with respect to the two lists of source files involved in the execution of each feature. The result of feature coupling queries is of similar format, but extended by feature identifiers.

Two architectural primitives are logically coupled if there is a logical coupling between the source files implementing the two architectural primitives.

Logical coupling between two recovered architectural primitives is computed using a query similar to the one described for the features. But instead of files affected by a feature two lists of files realizing the two architectural primitives are used as input to the query. Results of the query are of the same format as the feature data record, but instead of feature identifiers there are architectural primitive identifiers.

Concerning visualization of resulting views we apply various techniques ranging from graphs in Rigi or more advanced such as multi-dimensional scaling. For both techniques there is reasonable workaround to provide the data in proper format. We implemented this workaround in a number of scripts. We briefly describe latter technique in the case study. For a more detailed description concerning visualization of evolution data we refer interested reader to our previous work described in work [11].

5. Case Study: architecture evolution of Mozilla

In the following we demonstrate our approach on hand of an architectural evolution analysis task that we performed on the Mozilla open source software project (Version 1.3a). In our analysis task we concentrated on the browser feature

of loading a web site via a secure network connection (i.e. HTTPS). In particular, this feature addresses two important architecture aspects of Mozilla: communication and security.

5.1. Step 1: Evolution data preparation

Before starting the analysis we had to prepare the spaces of information. With respect to the architecture recovery space we applied Imagix-4D [18] to parse the source code to source models. Because of the size of the Mozilla we parsed each base directory separately and used the project composition facility of Imagix-4D to aggregate projects to super-projects. In addition to use the relational algebra tools Grok [9] for more complex filtering and aggregation tasks we exported the data to Rigi Standard Format (RSF). Table 1 lists the contents of the source model database concerned with Mozilla’s secure communication.

Table 1. Content source model database - Mozilla secure communication

Files (C/C++, Headers):	1.543
LOC (source, comments):	642.336
Classes (classes, templates, structs, unions):	2.068
Functions (defined, library):	18.732
Variables (local, non-local):	59.107
Macros:	10.197

For the release history space we retrieved the versioning data from the Mozilla CVS repository and the bug report data from the Bugzilla [38] bug tracking system and populated the release history database (RHDB) [13].

To validate the relationship between modification report (MR) and problem reports (PR) data we have used information available with PRs and secondary information such as patches. The following quantitative analysis shows that we achieved a very high success rate – thanks to the code maintainers for recording the IDs – in reconstruction of this information so data can be used in further analysis steps. Table 2 summarizes the results for PR data with a defined

Table 2. Problem reports of Bugzilla

\ Status Resolution	closed	resolved	verified
	all / ref	all / ref	all / ref
duplicate	282 / 5	13855 / 66	41648 / 487
fixed	333 / 63	14806 / 7705	36620 / 18940
invalid	315 / 6	4551 / 43	9116 / 113
won't fix	92 / 1	1823 / 46	3123 / 69
works for me	359 / 2	9765 / 93	15569 / 308

value in the *Resolution* field. From reports with status *resolved* or *verified* we found more than 50% in the important resolution-category *fixed*. Another important aspect is that

we found only a few reports with resolution *duplicate*, *invalid*, *won't fix*, or *works for me* which indicates an error rate of less than 1%. Since reports with status *closed* were not available in a large quantity – only 0.1% of all reports had this state – we omitted an evaluation. Table 3 depicts a

Table 3. Products and problem reports

Product	fixed reports			all reports		
	all	found	%	all	found	%
Browser	35520	20396	57.42	129889	22208	17.10
Bugzilla	1630	9	0.55	4564	26	0.57
MailNews	7089	4583	64.65	29112	4978	17.10
Tech Evangelism	1314	6	0.46	4908	16	0.33
mozilla.org	1126	5	0.44	2122	15	0.71

subset of the available products and summarizes our results with respect to product categories they were assigned to PRs. Column one labeled *fixed reports* list the number of PR whose state were set to *fixed*. From all available reports we found 57.42%. In contrast to all possible reports – column labeled *all reports* – we didn’t find significant more PRs. This can be justified with the argument that there should be no MRs for PRs which have been resolved as *invalid*, *works for me*, etc. Another product category we expected to find PRs is *MailNews* since it is part of the *Mozilla* browser project. The results are similar compared to *Browser*. For other products and discussion topics such as *Bugzilla*, *Tech Evangelism*, or *mozilla.org* we didn’t expect to find many reports. This may indicate a false positive detection but the rate is less than 1%. The building process and its results are discussed in more detail in [12, 13]

5.2. Step 2: Architectural primitive extraction

For the extraction of the communication and security specific architectural primitives we started with the determination of hot-spots in the source code that indicate the primitives. Design or code documentation often provides rich source for such characteristic statements. In case of the Mozilla’s secure network connection there is a separate project Network Security Services (NSS) that provides good online documentation. For our task we investigated the document “Overview of an SSL Application” [26] from which we obtained two hot-spots for opening a SSL connection (original text):

- `PR_NewTCPSocket`. Opens a new socket. A legal `NSPR-footnote socket` is required to be passed to `SSL_ImportFD`, whether it is created with this function or by another method.
- `SSL_ImportFD`. Makes an `NSPR socket` into a `SSL socket`. Required. Brings an ordinary `NSPR socket` into the `SSL library`, returning a new `NSPR socket` that can be used to make `SSL calls`. You can pass this

function a model file descriptor to create the new SSL socket with the same configuration state as the model.

Starting with the extraction of the communication primitive we queried the source model database for instances of the `PR_NewTCPSocket` statement. Using graph navigation and browsing facilities of *Imagix-4D* (and additionally of *Rigi*) we sliced the source model graph with respect to directly and indirectly linked entities of interest. Such entities are functions/methods that invoke or are invoked by `PR_NewTCPSocket`, and classes, files and directories related to these entities. Regarding the security primitive we concentrated on the hot-spot `SSL_ImportFD` and proceeded in the same way. The result is a subgraph of the source model containing entities that realize the secure communication of Mozilla.

However, the initial graph showed a vast amount of entities and relationships. To get a more clearer picture of the architectural primitives we applied the relational algebra tool *Grok* to aggregate information to the level of files, and respectively directories. The output of the aggregation represents a more condensed graph that is shown in Figure 3.

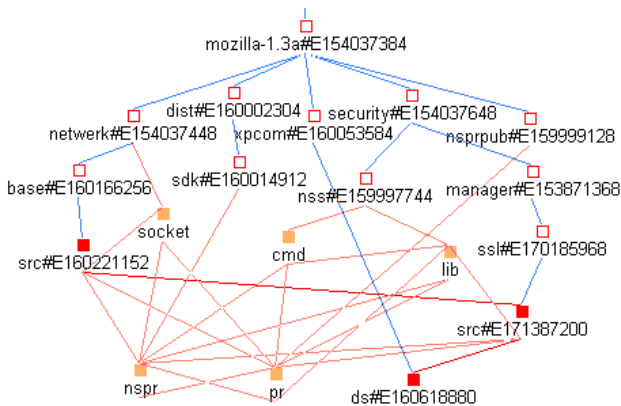


Figure 3. Mozilla secure communication.

The directory structure of Mozilla’s source repository is depicted top down from starting with the root directory `mozilla-1.3a` (the name of each node is followed by its unique identifier generated by the *Imagix-4D* parser). Subsequent directories represent the main parts involved in the realization of the communication and security primitive. Colored nodes are sub-directories that contain the files implementing the secure communication primitive. The most interesting main directories are `network`, `security`, and `nsprpub`. The interfaces of the `network` module that among other things are responsible for the establishment of a connection are implemented in the `base` directory of `network`. They access the higher-level socket services implemented by files of the `socket` directory which

themselves use basic input/output services provided by the `nsprpub` module. The very basic service is concerned with opening of a socket connection by the `socket()` statement.

The security part is implemented as library named `lib` that is a subdirectory of the `security` module. The services of the `network` module access security services of the library via the `manager` module. The dependency from `pr` to `lib` is quite unexpected, because `pr` implements basic services. A detailed analysis yielded that `pr` contains header files that declare functions, for example for monitoring, that are implemented by the security library. Hence, calls of these functions to other security related functions result in this unexpected dependency. The high number of references to the `nspr` directory is explained by the fact that `dist` contains the header files generated during compile time that are included by the other files.

5.3. Step 3: Feature analysis

Table 4. Scenario definitions

Scenario	Description
Core	mozilla start / blank window / stop
HTTP	TrustCenter.de via HTTP
HTTPS	TrustCenter.de via SSL/HTTP
File	read TrustCenter.de from file
MathML	mathematic in Web pages
PNG	sample image
XML	XML Base
JPG	JPEG Karlsruhe

The GNU tools [1] were already used successfully in [8] to extract feature data. Following we describe the basic steps of our feature extraction process on hand of our feature analysis of Mozilla.

We first created a single statically linked version of *Mozilla* with profiling support enabled. From several test-runs where the defined scenarios (see Table 4) were executed, we created the call graph information using the GNU profiler. The call graph information again is used to retrieve all functions and methods visited during the execution of a single scenario. Since our analysis process works on the file level, we mapped function and method names onto this higher abstraction level. In the next processing step, “fea-

Table 5. Extracted features

Feature	Color	Files
Core	White	705
Http	DeepPink	28
Https	MediumGreen	6
MathMIEExtension	YellowGreen	13
ImagePNG	DarkOrange	10
Xml	MediumOrchid	65
ImageJPG	Cyan	16
Html	DeepSkeyBlue	76

ture data” was extracted from file name mappings using set

As result we obtain a graph where nodes with high coupling are placed closer together compared to nodes with lesser coupling. The broadest lines in Figure 4 indicate five common reports, medium sized lines three or four reports, and thin lines one or two reports. The shaded area indicates the location of the *security concept* together with the features *Http* and *Https* which is clearly separated from areas with other features. Easy to grasp is the strong coupling between parts of network and security. Other features such as *Html* or *Xml* are more spread and also have a more complex coupling structure.

Detailed information about the process of visualization the coupling between features can be found in [10].

6. Related work

Mozilla has been already addressed by several research groups, for example, by Mockus, Fielding and Herbsleb in a case-study about Open Source Software projects [25]. In that work, they 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 [2] Bianchi et. al studied the entropy of a software system to assess its degradation by applying the entropy class of metrics on successive releases.

Wong et. al [37] 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.

In [23] 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 [17] 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 our previous work we addressed the issue of using version data for detection of logical coupling between of source modules and sub-systems. Particularly, we concentrated on the identification of change sequence patterns indicating files that are changed in a common way and thereby coupled [14]. Future work is on integrating this techniques into our architecture evolution analysis approach.

7. Conclusions

In this paper we presented our architecture evolution analysis approach that integrates the information spaces: 1) architectural primitives and system features identified in source models; 2) evolution data extracted from version history such as bug reports and modification reports. The discovered architectural primitives point to source model structures such as classes, functions and files; evolution data analysis reveals logical couplings between files (or finer grained source structures).

Our approach can combine these information spaces and allows tracing from architectural primitive to their corresponding evolution history and, thus, highlight particular architectural developments over time. Such analysis can be performed for different kinds of architectural primitives (e.g., communication, security, control, etc.) and, therefore, provide certain views onto a system. The engineer can use such views to reason about architectural shortcomings or design erosion.

Future work is concerned with the development of an architecture analysis workbench that further integrates the set of tools and techniques presented. In this context we also plan to consider related research tools such as Code-Crawler [6] that provide additional analysis and visualization capabilities.

References

- [1] GNU's Not Unix! - the GNU Project and the Free Software Foundation (FSF). <http://www.gnu.org/>.
- [2] A. Bianchi, D. Caivano, F. Lanubile, and G. Visaggio. Evaluating software degradation through entropy. In *7th International Software Metrics Symposium*, November 2001.
- [3] G. D. Birkhoff. *Lattice Theory*. American Mathematical Society, 1967.
- [4] B. W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

- [5] 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>.
- [6] Codecrawler. <http://www.iam.unibe.ch/lanza/codecrawler/>, 2003.
- [7] cppx: Open source c++ fact extractor. <http://swag.uwaterloo.ca/cppx/>, July 2003.
- [8] 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.
- [9] H. Fahmy and R. C. Holt. Software architecture transformations. In *Proc. of the International Conference on Software Maintenance*, pages 88–96, San Jose, CA, October 2000. IEEE Computer Society Press.
- [10] M. Fischer and H. Gall. MDS-Views: Uncovering of Coupling between Features via Problem Report Grouping. Technical report, Distributed Systems Group, Technical University of Vienna, 2003.
- [11] M. Fischer and H. Gall. MDS-Views: Visualizing Problem Report Data of Large Scale Software using Multidimensional Scaling. Technical report, Distributed Systems Group, Technical University of Vienna, 2003.
- [12] M. Fischer, M. Pinzger, and H. Gall. Analyzing and Relating Bug Report Data for Feature Tracking. In *10th Working Conference on Reverse Engineering (WCRE)*, Victoria, Canada, November 2003.
- [13] 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.
- [14] 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.
- [15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass. and London, 1995.
- [16] B. Ganter and R. Wille. *Formal Concept Analysis*. Springer, 1999.
- [17] 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.
- [18] Imagix-4d 4.2. <http://www.imagix.com>, July 2003.
- [19] M. Jazayeri, A. Ran, and F. van der Linden. *Software Architecture for Product Families: Principles and Practice*. Addison-Wesley, Reading, Mass. and London, 2000.
- [20] R. K. Keller, R. Schauer, S. Robitaille, and P. Pagé. Pattern-based reverse-engineering of design components. In *Proc. of the 21st International Conference on Software Engineering*, pages 226–235, Los Angeles, USA, May 1999. IEEE Computer Society Press.
- [21] C. Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proc. of the 3rd Working Conference on Reverse Engineering*, pages 208–215, Monterey, CA, November 1996. IEEE Computer Society Press.
- [22] J. B. Kruskal and M. Wish. Multidimensional Scaling. *Quantitative Applications in the Social Sciences*, 11, 1978.
- [23] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of IWPSE 2001 (International Workshop on Principles of Software Evolution)*, 2001.
- [24] N. R. Mehta and N. Medvidovic. Composing architectural styles from architectural primitives. In *Proc. of the 22nd International Conference on Software Engineering*, pages 178–187, Limerick, Ireland, June 2000. ACM Press.
- [25] 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.
- [26] Mozilla: Network security services (nss). <http://www.mozilla.org/projects/security/pki/nss/>, 2003.
- [27] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proc. of the 24th International Conference on Software Engineering*, pages 338–348, Orlando, FL, May 2002. ACM Press.
- [28] Per Cederqvist et al. *Version Management with CVS*, 1992. <http://www.cvshome.org/docs/manual>.
- [29] M. Pinzger, M. Fischer, H. Gall, and M. Jazayeri. Revealer: A lexical pattern matcher for architecture recovery. In *Proc. of the 9th Working Conference on Reverse Engineering*, pages 170–178, Richmond, Virginia, October 2002. IEEE Computer Society Press.
- [30] M. Pinzger and H. Gall. Pattern-supported architecture recovery. In *Proc. of the 10th International Workshop on Program Comprehension*, pages 53–61, Paris, France, June 2002. IEEE Computer Society Press.
- [31] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Vol. 2*. John Wiley & Sons, 2000.
- [32] Software Composition Group, University of Berne. *The FAMIX 2.0 specification*, 2.0 edition, August 1999. <http://www.iam.unibe.ch/scg/Archive/famios/FAMIX/>.
- [33] Source-navigator 5.1. <http://sourcnav.sourceforge.net>, June 2002.
- [34] S. R. Tilley, K. Wong, M.-A. D. Storey, and H. A. Müller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994.
- [35] 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.
- [36] 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.
- [37] 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.
- [38] Bugzilla Bug Tracking System. <http://www.bugzilla.org>.