Dissertation

EvoZilla - Longitudinal Evolution Analysis of Large Scale Software Systems

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Doktors der technischen Wissenschaften

unter Leitung von

Univ.Prof. Dipl.-Ing. Dr. techn. Harald C. Gall
Software Engineering Group
Department of Informatics
University of Zurich

und

o.Univ.-Prof. Dipl.-Ing. Dr.techn. Mehdi Jazayeri
Distributed Systems Group
Vienna University of Technology

von

Michael Fischer
Gabelsbergerg. 3/15
2700 Wr. Neustadt
Matrikelnummer: 8827769

Zürich, Dezember 2006

Kurzfassung

Langlebige Computerprogramme repräsentieren wichtige Vermögenswerte, welche möglichst flexibel auf die sich beständig ändernden Bedürfnisse ihrer Anwender reagieren müssen. Konsequenterweise muss die Prüfungen auf Unregelmäßigkeiten in der Entwicklung eines Computerprogramms ein integraler Bestandteil des gesamten Lebensdauerzykluses sein. Die Entwicklungsanalyse von Computerprogrammen offeriert eine Möglichkeit die strukturelle Stabilität während der Programmentwicklung zu bewerten. Erforderlich ist daher eine Methodik um kritische Programmartefakte noch während der Entwicklung des Systems zu identifizieren und mögliche strukturelle Mängel aufzuzeigen. Die so gewonnenen Informationen repräsentiert Rückmeldungen über die getroffene Entscheidungen in der Entwurfsphase und ermöglicht so deren Validierung.

Um das Problem der Informationsgewinnung zu lösen, konzentrieren wir uns in der gegenständlichen Arbeit auf das Potential historischer Daten welche automatisch während der Entwicklung eines Systems aufgezeichnet werden. Primäre Forschungsziele sind daher: (G1) ein effizientes Speichermodell und Werkzeuge welche die Speicherung und Auswertungen historischer und struktureller Daten ermöglichen; (G2) eine schnelle und effiziente Methodik zur Detektierung strukturelle Anomalien basieren auf historischen Daten; und (G3) eine Methodik zur Generierung von Feedback über die wichtigsten Artefakte und deren Abhängigkeiten um das Verständnis über Änderungen und deren Effekte zu verbessern.

Unsere Analyseansatz, *EvoGraph* genannt, dient zur Erkennung von kritischen Artefakten und Mustern im Zuge der Evolution eines Programmsystems basierend auf Änderungs- und Problembeschreibungen, Laufzeitdaten und Quellcodeänderungen. Zur effizienten Anwendung unseres Ansatzes werden die zu analysierenden Teile mittels Merkmalsanalyse vorselektiert. Basierend auf der gewählten Untermenge, wird ein Abhängigkeitsgraph erstellt, welcher Gewichtungen aus den historischen Informationen enthält. Mittels Distanzoptimierung können komplexe Abhängigkeiten zwischen den Knoten im Graphen dargestellt werden. Als Resultat werden stark voneinander abhängige Knoten in enger Nachbarschaft zueinander plaziert. Eine nachfolgende Analyse der Änderungen im Quellcode gibt detailiert Auskunft über die Entwicklung des Systems hinsichtlich struktureller Abhändigkeiten.

Ergebnis unseres Ansatzes ist eine qualifizierte Untermenge von ausgewählten Quellcodeartefakten gemeinsam mit ihren strukturellen Abhängigkeiten. Sie stellen das *evolutionäre Echo* der implementierten Architektur des Programmsystems dar. Darüberhinaus werden durch Visualisierungen entsprechende Informationen über Änderungsmuster im Quellcode geboten. Sie erlauben das Ziehen weiterer Schlüsse über die Entwicklung des Systems.

EvoGraph stellt einen flexiblen, einfachen und ausreichenden Ansatz dar um die strukturelle Stabilität von großen Programmsystemen zu überprüfen. Unsere Fallstudie mit der *Mozilla Application Suite* hat gezeigt, dass unser Ansatz praktisch durchführbar sowie effektiv im Aufzeigen der wichtigsten Artefakte und ihrer strukturellen Abhängigkeiten ist. Die Resultate zeigen weiters, dass langlebige Programmsysteme historische Daten in ausreichender Qualtität und Quantität liefern, um die effiziente Identifizierung von Anomalien zu erlauben. Hinsichtlich unserer Fallstudie waren wir in der Lage, *Gott-Klassen* in der strukturellen Dimension und verschiedene *Muster* in der entwicklungsmäßigen Dimension zu identifizieren. Sie sind ein Zeichen für strukturelle Instabilitäten welche komplexe und teure Quellcodeänderungen verursachen.

Abstract

Large and long-lived software systems represent important assets. With respect to their constantly changing and evolving environment they must react flexible to incorporate user requirements. Design for change and agile development are no panacea without appropriate feedback about the created instances of the solution space. As a consequence, the check for anomalies in a system's evolution must be an integral part of the overall live-cycle process. Software evolution analysis offers an opportunity to establish a feedback loop for the assessment of the structural stability of a software system. A methodology is required to systematically identify critical entities in a system's evolution with respect to their structural dependencies and to point out their shortcomings. This information represents the feedback about *and* enables the validation of past design decisions.

To solve the problem of information extraction and feedback generation we focus on the large amount of historical data which are automatically recorded by the various supporting tools such as version control or problem reporting systems. The primary research goals which arise are therefore: (G1) an efficient storage model and support tools which enable the accommodation of the historical and structural changes for fast and efficient analysis; (G2) a fast and efficient method to detect structural anomalies from release history and problem report information; and (G3) a method for feedback generation about critical entities and dependencies to improve the understanding about changes and their effects.

Central element of all analyses is a release history database which accommodates all data gathered about a system. We propose *EvoGraph*, an approach for hot-spot and change pattern detection in the evolution of large-scale software systems based on modification and problem reports, run-time data and source code changes. For the efficient use of our *EvoGraph* approach, we first apply feature analysis to obtain source file candidates for a detailed exploration. Based on the obtained file-set, a dependency graph is generated from the historical and structural information in the database. Next, the graph is used to reveal non-obvious structural dependencies, which is achieved via minimizing the distance between related nodes and maximizing the distance between unrelated nodes. Finally, a subsequent source change analysis phase provides detailed information about the system's evolution with respect to structural changes.

Result of the approach are qualified subsets of the pre-selected source artifacts with their respective structural dependencies, which can be conceived as the *evolutionary echo* of the as-implemented architecture. Structural changes are also quantifiable which facilitates their assessment on a fine-grained level such as method or variable. Moreover, visualizations provide feedback to draw further conclusions about identified change patterns and re-engineering intervals.

EvoGraph represents a flexible, lightweight, and sufficient approach to assess the structural stability of large software systems. Our case study with the *Mozilla Application Suite* has shown its applicability and effectiveness in pointing out the major artifacts and their structural shortcomings. Results also indicate that long-lived software systems provide historical data in sufficient quality and quantity to efficiently identify and point out structural and evolutionary anomalies. With respect to our case study we were able to identify *God-classes* in the structural dimension and different *anti-patterns* in the evolutionary dimension which are indicators for structural instabilities causing complex and costly source code changes.

Acknowledgment

The work described in this thesis was supported in part by

- the Austrian Ministry for Infrastructure, Innovation and Technology (BMVIT);
- the Austrian Industrial Research Promotion Fund (FFF);
- the European Commission in terms of the EUREKA 2023/ITEA projects CAFÉ and FAMILIES;
- the European Software Foundation under grant number 417; and
- the Swiss National Science Foundation (SNF) and the Hasler Foundation Switzerland.

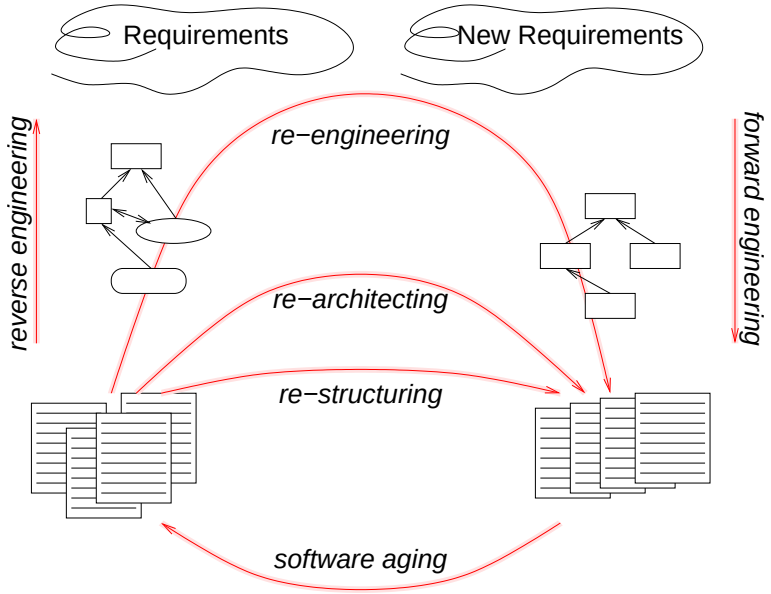
Contents

Chapter 1

Introduction

Changing requirements and technologies are the driving forces in the evolution of software systems. This observation is reflected by Lehman's first law about software evolution [95]—the law of *Continuing Change*—which states that software has to evolve to maintain its usability. As a consequence, these driving forces of software evolution can necessitate a complete redesign. Thus, software engineers are interested in having a clear picture about the agility of their systems. Evolution data can be used as basis for a decision whether and when a redesign should be made. Without proper maintenance activities software will age [110] (see also Figure 1.1) and will become less valuable for its users. Furthermore, real world experience indicates, that the development process of long-lived commercial software and open source software systems frequently exhibit a chaotic characteristics rather than a structured development process with well-defined phases. Undesirable results of a "stochastic" system development processes are sub-optimally structured systems with dependencies inducing ripple effects [145], requiring large source code changes sometimes affecting hundreds of files located in different modules, or causing difficulties in following their architecture blue-print in the course of their evolution.

Figure 1.1 Software maintenance Sisyphus: reverse engineering, forward engineering, software aging.



1.1 Motivation and problem description

Following the observations of Gall et al. [62] these shortcomings of a system's structure leave detectable traces in the development history. They pointed out that simultaneous changes extracted from the version history of software systems are strong indicators for logical dependencies between different sub-systems. These systems contain a vast amount of information about the reasons for small or large changes to the software, for which adequate data filtering mechanisms need to be applied to enable useful and meaningful analyses. Hidden dependencies of structurally unrelated but over time logically-coupled files (i.e., files which are frequently changed together although residing in separate modules or subsystems) further exhibit potential to illustrate structural evolution and possible architectural shortcomings. In recent research, simultaneous source code changes were independently used by Zimmerman et al. [149] and Ying et al. [146] with varying success to predict source code changes by mining co-change information. Especially the approach by Zimmerman showed that a pure statistical model delivers only unsatisfying results.

Mining approaches for historical information are frequently restricted to draw colored diagrams about various evolutionary and structural aspects such as identification of stable phases, distribution of problem reports, visualization of code metrics and others [45, 64]. Still, many of the recent approaches only inadequately provide clues about structural shortcomings and stability of a software system. Collberg [38] for instance proposed a visualization approach to trace a system's evolution on the basis of (call-, inheritance-, control-flow-, etc.) graphs with a temporal aspect. Shortcomings of a system are not indicated. Though Wu's approach [142] is designed *to highlight conspicuous changes across a historical sequence of software releases* is does not consider structural information.

A large amount of research work has been pursued in the past years with respect to architecture recovery [51], re-factoring and re-engineering [109, 41]. Pure structural detection approaches have shortcomings in pointing out relevant entities of ill-structured systems or constructs which harm a system's evolvability. Reason is that these approaches have no effective notion about the relevance or importance of the relationships between structural entities [86, 75, 120].

An exception is the work of Lanza who proposed the *Polymetric Views* [94] to combine structural and evolutionary metrics in the analyses. Usually, the interpretation of the various extracted source code metrics falls into the responsibility of an experienced system engineer. Source code metrics such as fan-in and fan-out, number of public methods and variables provide some bases for the building of candidate-sets of relevant entities and to measure a systems evolution [128].

Drawback of these approaches is that also entities are considered which are not well-designed but never modified. Thus a predictor would be advantageous providing information about relevant entities and to rank the entities for re-structuring activities. Seldom or never changed entities of lesser pertinence can be neglected or at least their re-structuring can be postponed.

Another resource which has been exploited by researchers with respect to evolutionary changes are code clones in a system. A few approaches have been proposed to track for instance changes within a class. Van Rysselberghe et al. [137] use clone detection technology to track clones between different versions to identify moved code or other anomalies in the source code. Other research approaches identify splitting and merging of classes due to re-structuring activities [18] or aim at providing a comprehensive picture about variations in code clones during system-evolution [89]. Though there have been interesting results with respect to particular clones or classes, an exploration [104] concerning the software system used in our case study did not indicate much potential for information which could be exploited to point out structural shortcomings.

Therefore, an approach is required for the extraction and evaluation of evolutionary and structural information to obtain a holistic view which can improve the understanding of a system's evolution and as a consequence bridge the gap between evolutionary and structural analyses. This has been realized for instance in the *ArchView* approach proposed by Pinzger [111]. Both aspects are considered to derive quality metrics on a high abstraction-level about the implementation of a software system. This is achieved via abstraction from the detailed to the higher levels via algebraic manipulations [79]. An analysis of architectural stability [132] has not been pursued. Since he also followed the "traditional" approach for fact extraction from the source code via regular parsers, an in depth analysis of evolutionary changes is not possible. Due to the high effort for parsing the huge amount of source code information, only a subset of the source files can be examined. The monitoring of ongoing projects seems to be difficult as well, since

always complete sets of the source code have to be evaluated.

1.2 Approach

EvoGraph is our proposal for the assessment of structural dependencies based on the evaluation of historical information. We aim at providing feedback about *and* enabling the validation of *past design decisions*. An institutionalization of our approach for the continuous monitoring of ongoing projects provides a just-in-time delivery of analyses results for the reasoning about structural entities. To support the reasoning process, a model of the software system is created where elements of the source code are related to elements of the evolution process. So architectural and evolutionary information can be provided for analyses on different abstraction levels.

This requires the integration of architecture and evolution into a single unified information space. Thus, a comprehensive database for evolution analysis on structural level needs to be populated with data from the information spaces release history, source code configuration, problem reporting and conceptual data as well. *EvoGraph*'s outcome are pointers onto structural entities together with their evolutionary patterns and their shortcomings. These entities should be considered as first class candidates for re-structuring or re-engineering activities to enable the incorporation of present and accommodation future user requirements.

Next, we provide a brief introduction to the topic of software evolution analysis, give definitions to facilitate the discussion, and describe our process model, the major challenges and the risks of our methodology.

1.2.1 Software evolution analysis

Retrospective software evolution analysis is concerned with the analysis of historical data of software projects and systems. Large amounts of this data are collected during the lifetime of software projects and stored in version control systems and bug tracking systems. Since this information describes various aspects of the evolutionary changes of software systems, they are a valuable source for a retrospective analysis. The results constitute views that characterize the evolution of a system and reveal critical points (hot-spots) in its design. Consequently, we define *software evolution analysis (SEA)* as *the investigation of a software system's evolution* **to identify potential shortcomings in its architecture** *or* **logical structure**. Tahvildari et. al [128] proposed to divide software evolution analysis into the following three categories: (a) *interface evolution* which relates to source code changes that affect the interfaces between functions, modules, or subsystems; (b) *implementation evolution* which relates to source code changes that affect the control flow and data flow properties of a given code fragment; and (c) *structural evolution* which relates to changes that affect the structure of the system but do not not necessarily affect its functionality, control flow, or data flow properties. Due to the information available, in this thesis we will restrict ourselves onto the third type of evolution.

The result of successful maintenance activities is the system's evolution. It establishes a strong relation between software evolution analysis and software maintenance activities. In [12] a standard for an iterative process for managing and executing software maintenance activities is described. Besides the definition of an appropriate vocabulary, this standard prescribes requirements for process, control, and management of the planning, execution, and documentation of software maintenance activities. Based on the vocabulary a closer integration of the maintenance activities with the respective repositories for instance to document the type of change such as corrective, enhancement, or perfective, would be beneficial for the software evolution analysis process.

1.2.2 Definitions

1.2.2.1 Longitudinal observations

In [7], *longitudinal* is defined as *involving the repeated observation or examination of a set of subjects over time with respect to one or more study variables*. Following the above definition, longitudinal does not imply that something has to change. This is in contrast to *evolutionary* which declares a process to be a

working out or *developing* process. As not all changes to a software system are of *developing* nature, we will use *longitudinal* in conjunction to activities, behavior, or views related with analyses of the system under inspection

Definition: We define *longitudinal* as *the timely or historical perspective an activity, a behavior or a view provides*.

1.2.2.2 Evolutionary hot-spot

Definition: We define an *evolutionary hot-spot* as *a structural entity or the relationship between two or possibly more entities with extraordinary characteristics in terms of the applied metrics capturing the longitudinal aspect and the average for this metrics computed for the system or part of it.*

Using a relative reference in the above definition has the advantage that variations between different systems or even within different modules of a single system do not bias the results. With respect to our approach which exploits the logical coupling between two entities, the identified hot-spots are those file-pairs which have been frequently modified together during the observation period. Other metrics which reflect evolutionary aspects of a single, isolated entity are for instance source code growing rate, source code change rate, change rate of fan-in and fan-out, etc. Such metrics are exploited in Lanza's *Polymetric Views* [94] to characterize class evolution. Though this approach is well-suited to identify files with problematic characteristics in their evolution, it does not provide a thorough insight into the evolvability of a set of files with possible mutual dependencies distributed over different modules.

Nevertheless, both types of hot-spots are frequently first class candidates for re-structuring or re-engineering activities. Metrics to identify single files are straightforward and implemented in a variety of flavors, e.g., lines of code, publicly accessible variables, cohesion, etc. The identification of file-pairs with evolutionary critical behavior is more difficult since the computational effort easily can reach $O(n^2)$. With tools such as Grok [78] based on Tarski's binary relational algebra [129] graphs can be manipulated in such a way that the search process can focus on the harmful entities. As this does not promise much advantage over our approach using a heuristics-based search we rely on our "conventional" strategy to identify the relevant entities. Besides the pure list of identified hot-spots we are also interested in investigating their grade of mutual dependencies as a network of nodes. In large long-lived systems these dependencies are frequently very complex and optimal clusterings are therefore difficult to find. Consequently, we apply visual graph layout algorithms to generate an appropriate feedback about the dependencies between the identified hot-spots.

1.2.2.3 Feature

The term *feature* has been used in the telecommunication domain for some decades before the software engineering community discovered the term for their purposes. A brief discussion about the definition of a software feature and feature interaction can be found in [116].

Definition: In accordance with the general definition of feature [7], *a software feature is defined as a prominent or distinctive aspect, quality, or characteristic of a software system or systems* [85].

The properties *aspect* and *quality* are very vague with respect to software systems. Besides the difficulties to measure and locate non functional aspects of a software system such as response time, reliability or maintainability, a software system is primarily designed to realize a number of functional requirements. These functional requirements—implemented as the behavioral aspects of the source code—are derived from the features requested by the users of the software system. Furthermore, our software evolution analysis approach is applied on source files and therefore we need a mapping of user features onto source code. Thus, we emphasize the behavioral property and adhere to the following more practical definition.

1.2.2.4 Software feature

Definition: A *software feature* is *an observable and relatively closed behavior or characteristic of a software part [116]*.

The problem which has to be solved in reverse engineering is to find a mapping of the *relatively closed behavior* onto source code. Since the implementations for different features may overlap a non-ambiguous

solution is not always possible. A technique which is known as software reconnaissance [140] gleans the required information via dynamic analysis from the executable software system.

The advantage of using features for analyzing software systems is that they are natural units in the communication between user and developer. Examples for features from the Internet browser domain are secure communication, the browser history or support for MathML. Furthermore, feature composition is used to build new features out of a set of other features. For instance the browser history feature is composed of forward and back buttons, a small database, a front end to modify the database, etc. A further advantage when considering a software system as composition of features is that, the relationship between features in terms of their evolutionary properties can be investigated.

The use of features in software evolution analysis can be further motivated by maintenance activities which become simpler when defect reports concerning a certain feature can be assign to designated source code files. Source code changes concerning interwoven features with unclear code boundaries may show unexpected side effects, e.g., in slowing down the system. Aspects [88] can be used to modularize features that otherwise would be scattered and tangled up with several components. Therefore, the use of aspects may minimize dependencies among components and between features as well. Also in product family engineering the notion of feature and clear code boundaries have advantages in composition new products and building a product family platforms. The features and their interaction in respect to the longitudinal perspective are subject to mining activities in software evolution analysis.

1.2.2.5 Interwoven software feature

Definition: We define *interwoven software features* as *a set of distinct software features which share a number of structural or logical dependencies*.

As outlined above, software features are not only valuable in the user-developer communication but also facilitate the understanding of the evolution process. Usually a feature is not located in single files rather they span across several source modules and also cross-cut several abstraction layers of a system's architecture. This intrinsic interwovenness of features, i.e., the degree of structural and change dependencies between a set of features, is a threat to the maintainability of a software system through unexpected side effects.

The relationship between different software features can be expressed, for example, as the proximity between them, based on the number of modification and problem reports that their implementing files have in common. Visualization of such interwoven features must emphasize the proximity between features so that hot-spots can be captured easily. These hot-spots indicate locations of design erosion, or even evolution on the architectural level. Reasoning about hot-spots requires support through traceability and zooming in on evolution and source model level. Otherwise, any conclusions about the cause of a hot-spot might be incorrect and lead to false decisions.

1.2.2.6 Change transaction

Definition: A *source code change-transaction* is a *timely coherent sequence of check-ins of several, not necessarily logically-related files into the source code repository*.

Notable for CVS [35, 73] there exist two sources to reconstruct these change transactions: (1) when logging is enabled then the time-stamp of the commit—which is the same for all files of a change transaction—can be used; (2) another option are the time-stamps when files are checked check-in into the repository which is recorded individually for each file. Though the first source would provide more accurate data—its availability depends on the configuration of the repository—we have to use the latter one in our case study to reconstruct the information.

1.2.2.7 Co-change

Definition: The term *co-change* refers to *files that participate in the same change-transaction* [19, 27].

Particularly file-pairs which are frequently co-changed are considered to have logical coupling (see next section). Naturally, the are of special interest for a further investigation via software evolution analysis.

1.2.2.8 Logical coupling

Definition: We refer to *logical coupling* as *two files or source code entities are logically-coupled if a modification to the implementation affected both source code entities over a significant number of releases* [62].

The logical coupling is determined from co-changes when considering the historical dimension of change transactions, whereas the *strength* of logical coupling between a selected pair of files is determined by the number of co-changes which occurred during a given observation period. The stronger the logical-coupling the higher should be the number structural dependencies. But, other causes such as code clones [34] are possible as well. Logical coupling itself makes no assumptions about the location of the files. We therefore introduce *stickiness* and *adhesion* to distinguish logical coupling of files which reside in different modules and logical coupling of files which reside in the same module.

1.2.2.9 Stickiness and adhesion

Definition: We define *stickiness* as *the number of cross-cutting change-transactions between any two file-pairs connecting two different structural units*.
Definition: We define *adhesion* as *the number of change-transactions between any two entities within a single structural unit*.

The definitions are required since for the determination of the logical coupling the location of the respective file-pairs is irrelevant. In the software engineering community, *coupling* and *cohesion* as coined by Yourdon [147] describe the relationship of *structural entities* (functions, variables, etc.) between and within higher structural units such as files, modules, etc. Furthermore, *module-coupling* and *module-cohesion* are already used to describe the architectural relationship between and within modules.

To avoid confusion with these terms, we use *stickiness* for cross-cutting change transactions (see also Figure 1.2) and *adhesion* for logical coupling of files within a module. Analogous to coupling and cohesion the stickiness should be low and adhesion should be high. This can be argued by the property that files within a module have more structural dependencies—presumed the system engineer follows common practice. To point out such possible design-shortcomings, our approach relies primarily on the *stickiness* property.

1.2.3 The EvoGraph approach

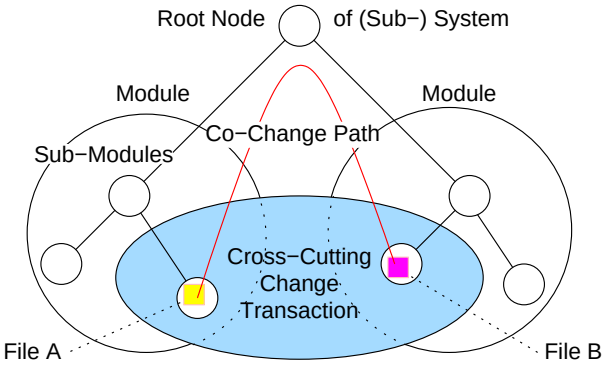
In this thesis we describe an efficient approach called *EvoGraph* to extract evolutionary and structurally relevant entities from modification reports—including source code analysis—to obtain a holistic view about the longitudinal development of a software system. Next, we provide a brief outline of our approach.

1.2.3.1 Data sources

Besides source code, design information and other development related information, large software systems contain huge amounts of historical data. This information needs to be filtered, assessed, condensed, possibly shifted to higher abstraction-levels, to gain the essence of a system's history. An important set of the historical data are modification reports which are recorded in source code repositories such as Concurrent Versions System [35, 73], Subversion [15], BitKeeper [11], or Rational ClearCase [5]. Though the systems are different, they basically record code deltas and their respective revision information. Usually several files have to be modified to add new functions or to fix a particular problem.

Another set of historical data are problem or bug reports. Source files modified due to problem reports can be related via the problem reports they have in common. This introduces a new quality aspect since problem reports are frequently related to certain features and users mainly think more in terms of features rather than software units. Problem reports are tracked via problem tracking systems such as *BugZilla* [13] or GNATS [9] published as open source software, or as commercial systems such as DevTrack [131] or JIRA [20],. Most open systems lack the integration of the problem tracking system into the version control system. Thus the link between these two repositories has to be established by other means.

Figure 1.2 Schematic representation of a cross-cutting change transaction involving two files from different modules.



1.2.3.2 Mining cross-cutting change transactions

These dependencies are subject for mining activities to provide starting points for further monitoring or appropriate re-engineering activities. Figure 1.2 schematically depicts a source code change comprising two files which propagates out of the well-defined module boundaries. The reasons for such change dependencies lie in yesterdays design decisions and implementations [69]. It is therefore useful to understand how such dependencies came into existence and how they evolved.

In current release history based mining approaches [27, 146, 148] only simultaneous changes of files are considered which has two major drawbacks: a large number of changes is not considered at all and local bias the results. Interesting results form our case study indicate

- that change transactions comprising a single file account for 42% of the source code changes in our case study;
- that more than 50% of the change transactions affect files which are located in the same directory; and
- that for the class of three files this property is valid for about 36% of the change transactions.

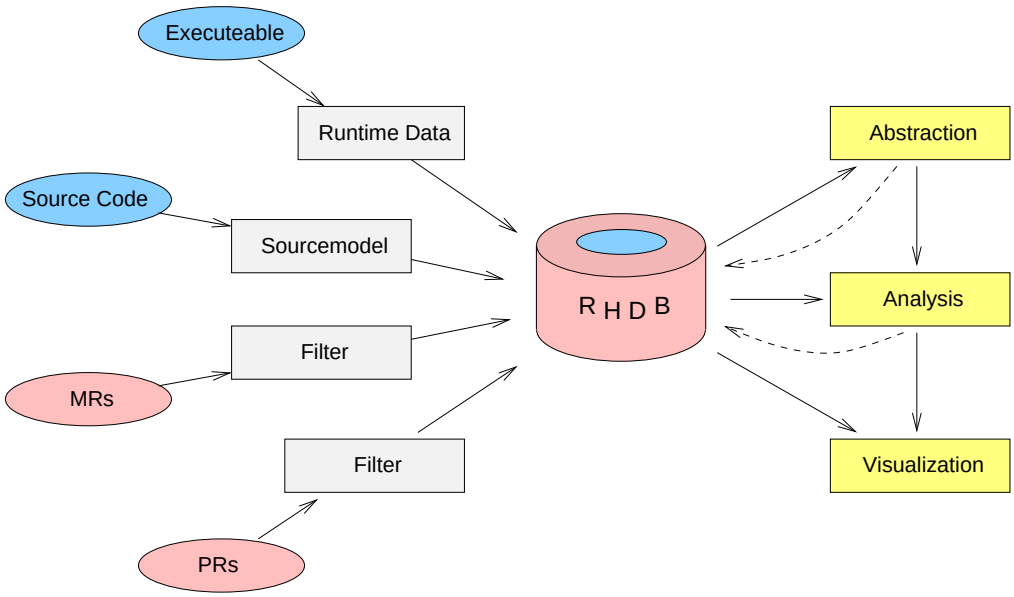
From another perspective, file-pairs that change frequently are most often located in the same directory. For example *.cpp* and the corresponding header files *.h* are often simultaneously modified. With respect to our case study, it is not a surprising result that file-pairs located in the same module have a three to four times higher change frequency than file-pairs which are located in different modules. Another observation is the number of files which are part of the same simultaneous change: it tends to be smaller for files which are placed closely within the module structure of the software system.

Though changes crossing module boundaries happen less frequently, they expose the system to large and complex modifications. For the purpose of validating major structural properties, we need to identify simultaneously changed file pairs on a global level. Since their frequency is lower than local changes, a simple ranking is not sufficient and the detection process therefore requires a higher computational effort.

1.2.3.3 Processing model

Additionally to historical information we use runtime data, source code information, conceptual and abstract information to enrich our model. This enables filtering of structural entities or reasoning about the causes of source code changes. Central element in all information processing stages is a relational database, depicted in Figure 1.3 as RHDB, which contains the data imported from the various sources, the intermediary results and the final data sets. Before evaluation can take place the gathered raw data has to be filtered, ambiguities have to be resolved, missing information has to be reconstructed, and implicit information has to be made explicit. This yields to an assured fact base which serves as input to the subsequent processing steps.

Figure 1.3 Scheme for processing information obtained from structural and historical sources.



Then, from the fact base the logical coupling between all artifacts are computed. The result is a cyclic, undirected, weighted graph of dependencies, whereas the weight of an edge indicates the degree of dependency. The higher the edge weight, the closer is their structural relationship.

Next, the selection of an appropriate set of artifacts narrows the search space for a further detailed evaluation to the architecturally relevant entities. In most cases artifacts representing the reference architecture will be chosen. Where this information is not available an approximation has to be made. The results from feature analysis are well-suited for this approximation, since it links the abstract concept of features with concrete entities such as files. The approximation is made by selecting a set of features which cover the major building blocks with respect to feature decomposition of the software system.

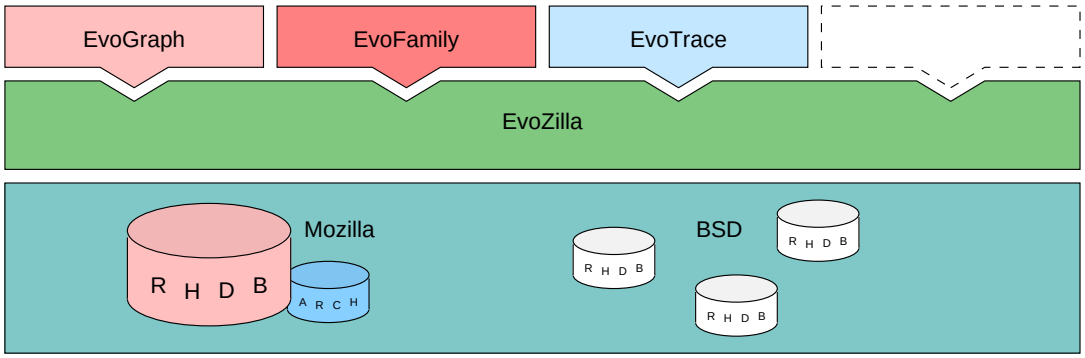
Based on the selected set of features or set of artifacts, a dependency graph is generated representing the structural map of entities. The goal of the following layout phase is to minimize the distance between related nodes and maximize the distance between unrelated nodes. A subsequent source change analysis phase models the longitudinal evolution. We use lightweight parsers for the various artifact types to identify those source code entities which are the cause for structural dependencies.

1.2.3.4 The EvoZilla framework

The *EvoZilla* framework is the vehicle to perform our evolutionary analyses and consists of about 135 Java files which represent more than 50,000 lines of source code. Additionally some Perl and Shell scripts are used to extract and import data from the different information sources such as version history or runtime data. Figure 1.4 depicts an architectural sketch of the framework. The analysis approaches such as *EvoTrace*, *EvoGraph*, and *EvoFamily* build the “application layer” and use resources of the framework which consists of:

- classes for commonly used data data-objects such as a global project time-scale, change sequences for the assessment of individual entities, project structure for distance measurements, cluster information, etc.
- persistency management classes: since we did not use a third party component such as Hibernate [2] for a persistence layer, we use some simple Java classes for the management of the content in the DB tables;
- analysis and visualization tools: a number of tools are available which are used for instance in the preparation phase of the *Release History Database* such as for the reconstruction of a time-scale or

Figure 1.4 An architectural sketch of the *EvoZilla* framework.



the co-change transactions. Other classes provide functionality for the representation of graphs and distance matrices or for drawing evolution diagrams; and

- launch pad: the analyses and report applications of the plug-ins are started via appropriate mechanisms provided by the framework. It provides database connectivity and generalized parameter handling.

The bottom layer is made up of a relational database system in our case MySQL [4] which hosts different *Release History Database*s about the systems which are investigated. Next to the *Mozilla Release History Database* the database with architectural information *ArchDB* is depicted which indicates the relation to the architectural reconstruction process *ArchView* [111].

1.3 Thesis statement

In this section we formulate the hypotheses for our research goals. Furthermore, we refine and discuss these goals based on the terminology introduced earlier in this section.

1.3.1 Hypotheses

Since the hypotheses build a chain of causality we provide also a “flow-chart” to depict their relationship and the impact on the overall acceptance and rejection. For the fulfillment of our research goals *G1*, *G2*, and *G3* we rely on the acceptance of the following hypotheses.

1.3.1.1 Correlation hypothesis

Hypothesis H1a: Logical coupling points out structural dependencies between source code entities.

Based on results from earlier studies [62, 63] we suppose that logical coupling between different files may point out structural dependencies as well. From different research studies [148, 146] and our own empirical studies [112, 114, 118] we can conclude that a strong relationship between logical coupling and structure in many software system exists. In our investigation about the correlation between historical information and a software system’s structure we were able to identify a number of causes for the occurrence of logical couplings:

- structural relationship: a direct structural relationship such as class inheritance, variable access, or method invocation cause frequent modifications of both participants. In our case study this is the most frequent cause for logical coupling;
- semantic relationship: unstructured data objects contain information which is processed by entities. Changes in the structure or semantic of the information requires to adapt the producer and the consumer;
- duplicated source code (code clones): structural and/or semantic code clones [104];

- administrative issues: source files are modified without changing the functionality of the code. A copyright license change is an example for such a modification;
- inconsistencies in the repository: duplicated records of entities in the repository due to manipulations indicate strong dependencies which do not exist.

We will prove this hypothesis with respect to the *Mozilla Application Suite* via empirical information extracted from our case study. On the basis of a structural study we will show for selected parts of the system the strong correlation between different types of structural relationships and logical coupling.

1.3.1.2 Traceability hypothesis

Hypothesis H1b: In the course of the evolution of a software system, structurally relevant entities leave traceable footprints in the system's history.

We assume that the decomposition of the system or the implementation of design patterns have some possible shortcomings, which are reflected in the logical coupling of particular file-pairs. This hypothesis is based on *H1a* and addresses the problem that entities in an ill-structured system have a higher likelihood to be modified as part of a co-change transaction than entities in well-structured systems. As a consequence, the logical coupling between certain entities will be higher. An example for such a shortcoming is a central event-handler routine. Without proper application of a design pattern, each new function has to be "hacked" into the event handler routine leaving traceable footprints. The application of the *observer pattern* (also known as publish/subscribe) could decouple the implementation parts and consequently reduce the required co-changes significantly. As already outlined, two types of logical coupling can be distinguished:

- (expected) logical coupling which occurs frequently within a module for instance between a **.cpp*-file and the corresponding **.h*-file; and
- (unexpected or harmful) logical coupling between entities located in different architectural units such as sub-systems or modules.

Since the latter one defines the *hot-spots* we are looking for, the metrics to apply and reveal these couplings depends on the structural properties of the system. Additionally, we use conceptual information such as module information or feature information.

If we cannot find evidence in our case study to support this hypothesis *H1b*, i.e., no relevant logical couplings can be found, the consequence under the precondition of the acceptance of *H1a* will be the rejection of *H1b* and the conclusion would be that the system is well-designed. As the result depends on the filtering strategy for the harmful logical couplings, the filtering strategy has to be selected carefully.

1.3.1.3 Stickiness filter hypothesis

Hypothesis H2: Clustering of system graphs based on filtered logical coupling information reveals structural relationships.

As outlined in the previous sections, *stickiness* is much smaller than *adhesion*. Thus, considering a "brut-force" approach for clustering of files [27] based on all logical couplings regardless of their type, will build groups of files stemming from the same module. This is not a surprising result since it basically reflects what is known a-priori. Though the more interesting entities with a high *stickiness* are placed in close neighborhood they are difficult to identify since the are frequently obfuscated by a number of less interesting entities. By interesting we mean an entity which acts for instance as a "peer-node" to other modules or have a high number of variable accesses from outside of the module. To suppress the impact of uninteresting logical couplings, the desired information must be selected first via an appropriate filter criterion such as structural properties or abstract concepts such as features.

1.3.1.4 Source diff hypothesis

Hypothesis H3: Source code analysis based on source code deltas obtained from version control systems provides sufficient evidence to characterize the structural dependencies with respect to their longitudinal development.

Information about source code changes are offered in two different flavors: (a) the full source code for every revision (or release); (b) the difference between two revisions (or releases). An approach to analyze the changes of several hundred revisions of a source file with a full-fledged parser is a computational expensive task and therefore unreasonable for a large software system. Since only small portions of the source code are changed frequently, most of the parsing work would be redundant. We therefore propose to use the source code changes available from the version control systems, which allow a fine-grained analysis of the changed portions. Then, structural dependencies can be extracted via a lightweight fact extraction approach. A possible risk arises from the missing structural context under which the source code is parsed and the difficulties to support namespaces.

1.3.1.5 Logical relation of formulated hypotheses

We presume that we are able to instantiate a specific process when the corresponding hypothesis is accepted. Consequently, the following flow-chart depicted in Figure 1.5 is the result of the logical relation of the hypotheses formulated earlier.

Hypothesis *H1a* builds the foundation and is therefore our starting point. Rejecting *H1a* implies that there is no correlation between logical coupling and structural information or just by chance. As a consequence, logical coupling could not be used as predictor. In the next logical step we validate *H1b*. Possible threats to its acceptance are a perfectly designed system in our case study, not enough information available from the system's history, or inappropriate filtering mechanisms. Rejecting $H1a \vee H1b$ implies that a detection approach based on historical information would not be an appropriate means to identify structural shortcomings.

Following, we evaluate *H2* against the combined evolutionary and structural information. The branch block in the diagram is therefore colored pink (▭) and blue (▭) to indicate the transition from the evolutionary aspect to the structural one. Rejecting *H2* means that we are not able to interpret the relationships between identified entities via clustering. Reasons for rejecting this hypothesis could be that the system is completely unstructured, the filtering mechanisms are not appropriate, or not sufficient historical information is available. The consequence for the *EvoGraph* approach is that no qualified, automatically extracted set of input data is available for the generation of structural feedback. In this case the information has to be provided manually.

H3 is independent of *H2*. But in our *EvoGraph* approach the data are further processed on the earlier results obtained and therefore this step relies on *H2*. Possible reasons for rejecting *H3* are the insufficient information which can be extracted from the source code deltas such as naming ambiguities or difficulties in correlating the results from the different programming languages.

The overall result in case of accepting $H1a \wedge H1b \wedge H2 \wedge H3$ will be therefore a success of our approach. As the approach may produce useful results if $H2 \vee H3$ have to be rejected, we claim a weak-success. In case of rejecting $H2 \wedge H3$ the approach produces no useful results.

1.3.2 Research goals revisited

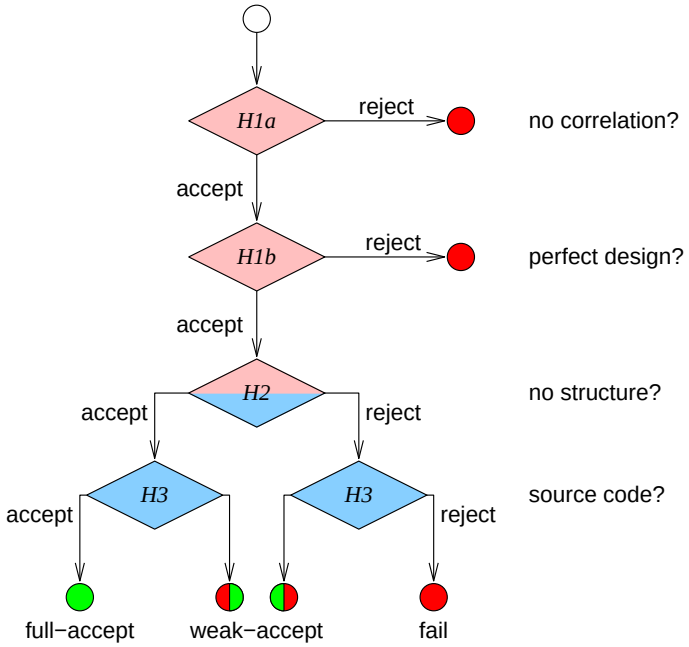
The research questions sketch the way how the overall objective of this thesis shall be fulfilled. We now refine and discuss the aspects of our research goals formulated earlier.

1.3.2.1 Storage and computational model

Goal G1: An efficient storage and a computational model which is able to accommodate the historical and structural changes for fast and efficient analysis.

The implementation of the intended approach must be able to handle several thousand files, and hundred-thousands of modification- and problem-reports. Consequently, we will use a relational database system to handle this amount of data efficiently. A pure file-based approach as it is common practice in architectural analysis with the FAMIX [123] model, has some major drawbacks such as long loading times and large memory requirements. Furthermore a specific model to store the evolutionary information does not exist so far. A comprehensive meta model for release history data including a computation model is still work in progress [124].

Figure 1.5 Logical relation of the formulated hypotheses building a “control flow”.



Also at the storage model the link with the architecture recovery process [111] is established. Since a number of results obtained are file-level based, though finer-grained (method-level) or coarser-grained (module-level) are possible. We will use file-level information to interconnect the evolutionary with the architectural models and processes, respectively.

Furthermore to avoid computational overheads, intermediary results will be also stored in the database. One of these results are for instance the co-change transactions and the respective files. Together with the problem report information they build the *qualified source change information*. Other information about the respective system under inspection are the reconstructed time-scale, source code metrics, or mappings of features onto source code.

1.3.2.2 Detection of structural entities

Goal G2: A fast and efficient method to detect structural relevant entities from release history and problem report information.

The implemented method will rely on the hypotheses *H1a*, *H1b*, and *H2* which pave the way for a fast detection of the relevant logical couplings. Large software systems frequently contain several thousand or even hundreds of thousands files and therefore a fast and efficient method for the detection of file-pairs which harm the system structure is important for the monitoring of ongoing projects.

Depending on the filtering strategy, relevant couplings for a further inspection may be for instance those which have been changed more recently than others. This method is similar to Gîrba’s *Yesterday’s Weather* [69]. It is an analysis method-based on the retrospective empirical observation that classes which changed the most in the recent past also suffer important changes in the near future. The approach of using historical information emphasizes the aspect of frequently modified structural dependencies. But also non-structural dependencies such as data dependencies can be pointed out, presumed that their semantic modifications also cause frequent changes in the source code.

Visualizations of the logical couplings with respect to their degree of dependency are one result of the implementation. A system expert can draw his conclusions about the as-implemented system with respect to the as-designed system. The identified hot-spots are candidates for re-structuring or re-engineering activities. In the realization of the next goal, the identified out entities are used as input for a further

investigation of their dependencies and their evolution.

1.3.2.3 Feedback generation

Goal G3: A method for the structural feedback about critical entities and dependencies of the system under inspection for a longitudinal development.

To fulfill the requirements for this research goal we will rely on *Source diff hypothesis (H3)* which is a consequence of the requirement to provide the results in a fast and efficient way. In the case of an acceptance of *H3* we are able to reconstruct the longitudinal development between selected entities and to detect evolutionary patterns in the evolution of structural dependencies. The idea is similar to Lanza's categorization of classes in his *Evolution Matrix* [93]. Furthermore, based on the extracted structural information, quality metrics will be established, which provide indications for the structural stability or instability of a part of a software system.

To remove the barriers between different programming and scripting languages used in the case study, some minimal meta-information about the implementation details are useful and necessary. Such meta-information are for instance the form of calls to components, naming conventions, or some basic information about the architecture of the system.

We use the following scenario as the motivating example: a button in a graphical user interface (GUI) is defined in an XML like description language. Some action from the underlying software system may be bound to this button. The question which shall be answered is, how this button in the GUI is coupled with the underlying software system and how the respective parts of the system evolved.

Though *G3* is as an independent research goal, we will use the results obtained via *G2* to instrument the information generation of this processing step in our case study.

1.4 Relevance, benefits and expected results

Our proposed *EvoGraph* approach offers a broad spectrum of results for the stakeholder through retrospective software evolution analysis. Furthermore, for the area of re-structuring or re-engineering our approach provides also novel insights with respect to a system's longitudinal development. In the following, we outline major characteristics of our approach and describe possible impact on related working areas.

1.4.1 Relevance

In our approach evolutionary information such as modification and problem reports or source code data from different releases are gathered and processed. One key result of the *EvoGraph* approach is the generation of detailed information about a system's longitudinal development with respect to its structural properties. The approach enables the reasoning about relationships as graphs with two different types of edges between entities: a historical and structural relationship. Following we discuss this characteristics of our approach with respect to its environment.

1.4.1.1 Stakeholders

Stakeholders of our approach are system engineers such as software architects, re-structuring and re-architecting experts, and other researchers in the area of software evolution analysis and architecture recovery. Due to time constraints of customer and market demands on ongoing projects an institutionalization of new system analysis methods is difficult. In these cases the retrospective analysis is useful to provide the stakeholders of the project with feedback about pros and cons of past design decisions.

While for the system engineers the generation of appropriate feedback will be the main interest, researchers with interests in architectural evolution might focus on the integration of the *EvoGraph* approach or its results into other analysis approaches. Thus, for other researchers also the detailed results of our case study are relevant for comparative studies. Therefore, the choice of a representative case study has to provide sufficient information to test and validate related approaches as well.

1.4.1.2 Product families

Although our *EvoGraph* approach is primarily designed for the exploration of a single large software system, the application onto a family of related products is enabled via its scalability and the capability to evaluate different types of historical couplings. For related systems or a product family it is even more important to ensure the agility of their platform architecture. A product family architecture can be evaluated as well via the detected and historically related file-pairs and allows one to reason about its potential to evolve and to accommodate future user requirements. This is of importance for system manufacturers with large software portfolios, since it is possible to detect deviations early and to react accordingly. To evaluate a set of related systems or platform architectures other dependency information than logical coupling are usable as well. For instance, common indicative keywords in the logs of the release history or function names similar in spelling can be exploited to recover interesting dependencies. This enables the detection of dependencies across release boundaries *and* product boundaries as well. Nevertheless, for all analysis types sufficient historical information in quality and quantity has to be available. The detected dependencies are exploitable with respect to providing of a holistic view about relevant structural dependencies between different systems or parts of them.

1.4.1.3 Interfacing

Interfacing the *EvoGraph* results with the structural analysis processes on the "natural" unit of file-level is possible and profitably for the assessment of the architecture of a software system. Furthermore, also on the finer-grained level of method and variable names, information about recurring patterns in their evolution can be exchanged with other approaches such as reverse- or re-engineering approaches by accessing directly the *Release History Database* or exporting the required information as RSF [141] data. The anti-patterns in the evolution of methods and variables point out possible instable structural relations. Metrics measuring the stability or in-stability of methods and interfaces provide additional clues for the structural reasoning process. For information exchange the exact reconstruction of a system's time-scale and released code versions is required. Otherwise results of various analysis processes are potentially difficult to synchronize since source code entities such as methods may have changed their physical position in a file due to source code changes.

1.4.1.4 Institutionalization

Possible effects on related research fields are the tighter integration of evolution analysis with existing structural analysis approaches. With the integration of the evolutionary information into the structural analysis models, the static characteristics of snap-shot models receive a dynamic component. This dynamic component is well suited for the institutionalization of prediction models for structural changes. In contrast to a pure retrospective analysis approach this combined analysis approach has the advantage of pointing out areas of a system with shortcomings requiring higher maintenance efforts in the near future. This enables the scheduling of re-structuring activities in accordance with the development strategy of the overall system.

1.4.1.5 Restructuring

A frequent required activity is the re-structuring or re-factoring [109, 60] of a software system during the development or maintenance phase to accommodate future requirements. In an empirical study [118] we successfully used logical couplings as indicators for structural shortcomings. Furthermore, it is possible to validate the effects of re-structuring activities via their further change behavior. Objective is in any case the improvement of a system's evolvability. Such improvements are for instance the removal of God classes via appropriate design patterns.

1.4.1.6 Version control systems (source code configuration)

Main data sources exploited in our case studies are version control and bug tracking systems. As these systems are primarily designed to track software configurations and problem reports, they have some limitations with respect to our requirements from the software evolution perspective.

One major shortcoming is the missing integration of the problem reporting system into the version control system we experienced with the used systems. Other (commercial) tools provide better integration. With respect to retrospective analysis a classification (new functionality, perfective change, bug fix, etc.) of the source code changes according to IEEE Standard for Software Maintenance [12] would be desirable. A classification ex-post is cumbersome and does not deliver adequate results. This information should be mandatory when committing changes to the repository.

Also some semantic and feature descriptions (interface change, new function, GUI change, etc.) would be advantages in a retrospective analysis process. Possible application fields are the semantic evaluation of structural dependencies and their changes. For our and other related analyses the provision of source code information such as identifiers with name-scopes of the changed source code elements would be beneficial. Though this is beyond the designed purpose of a version control systems, a mechanism to plug-in such concepts would ease such new developments.

Development platforms such as Eclipse [1] support already source code parsing. Consequently, a solution is required to store this meta-information in the repositories as well. The advantage would be that name-triggers could be instantiated and activation of predefined actions could take place. An action could be a warning message or the evaluation in structural dependencies with respect to new appearances of dependencies. Although some systems provide specific meta information such as source code patches, the detection of the correct association within the source code is difficult or even impossible. A formal mechanism to bind this meta information to source code changes frequently does not exist but would be required.

With respect to software evolution analysis, the tighter and standardized integration of the different information sources from the very beginning of a software project would ease the data preparation for a retrospective analysis and would be also beneficial throughout the whole development phase. Finer-grained information gathered during, e.g., editing sessions of source files, would contribute to the development of new analysis methods as well.

1.4.2 Benefits

Code required to realize a certain user required function or software feature may span across several modules of a system and therefore introduces possible harmful dependencies leading to “ossification” of the system. A typical usage scenario for the application of our *EvoGraph* approach is the generation of visual feedback about logically-related entities in large, long-lived software systems. Main task is the uncovering of relevant structural dependencies between selected entities based on release history information and in depth discussion on the basis of the generated visualizations. With the resulting mental model about the uncovered dependencies in mind, the adjustment of the system’s design and development process can be initiated and optimized.

1.4.2.1 Dependency evaluation

Via its computational model our *EvoGraph* approach facilitates the analysis of two indirect related information spaces such as temporal data from the system history and concrete data obtained from the system itself. Data from the temporal dimension such as the logical coupling are used to identify the important artifacts and relate them to each other. Data from the concrete dimension are for instance structural data with task oriented characteristics such as source code implementing a new feature. The benefit of this approach is that the metrics used to assess a system stem from the surrounding environment reflecting the changing requirements and other influences. Generally speaking, our approach exploits and filters commonalities based on data in one dimension—the logical dimension—to generated feedback about data in the other dimension—the structural dimension.

The uncovering of dependencies is performed in retrospective manner to direct the system expert’s attention to evolutionary hot-spots with outdated or inadequately design properties. As not all modules or parts of a system are equally important, criterion for pre-filter are for instance abstract concepts such as the core architecture, certain features, or more concrete entities such as modules. Subsequent analysis steps are based upon these hot-spots. Furthermore, they have to be considered as first class candidates for

re-structuring or re-engineering activities. The support for this advice is based on the evidence extracted from the release history.

1.4.2.2 Feature analysis

As already outlined features are natural units in the discussion with the users of a system. Therefore, a central concept of our analysis approach is the application of software feature information in the analysis process. From the software engineering and maintenance perspective, interwoven software features are undesirable results of the orthogonal characteristics of abstract concepts and concrete realizations. Moreover, in feature composition larger features are built out of other, smaller features. On the other hand, with composed larger features the likelihood increases that structural concepts such as module boundaries are violated. If this violation represents a significant structural harm and are therefore exposed to source code changes, the resulting frequent changes are recorded in the systems history and can be detected via our *EvoGraph* approach. Thus, the determination of the interwovenness provides an important feedback about the number and kind of dependencies between different features. The evolutionary and structural analysis indicates expected or unexpected dependencies. Whereas the latter, surprising dependencies may be responsible for unexpected side effects. Though not all dependencies are resolvable via appropriate re-structuring activities, the analysis results provide relevant information for the feature engineering process and the maintenance phase of a software system to create awareness about possible change impacts on other features. The systematic detection and indication of structural shortcomings between features are also results of our approach.

1.4.2.3 Structural shortcomings, evolution tracking, and patterns

The integration of different information spaces, such as history and structure, is a prerequisite for the reasoning about structural shortcomings, stability and maintainability of a software system. As indicated, our approach bridges the gap between these two spaces and therefore provides a novel level of quality for feedback. By exploiting these two information space, the longitudinal characteristics of structural dependencies between selected sets of entities is reconstructed. Since planned and unplanned structural structural changes leave marks in both of the above mentioned information spaces, we can reconstruct their longitudinal development via the analysis of source code deltas provided by version control systems.

The advantage of our approach is the possibility to track the evolution of a software systems structural dependencies on coarse- and detail-level as well. While coarse-level results inform about evolutionary dependencies such as modules or sub-systems, the detail-level information e.g., on method or variables, provides the required insight how the different larger entities are structurally coupled. The benefit for the system analyst is that s/he receives detailed feedback about the systems agility with respect to the implementation of different requirements. Focus of such detailed structural analyses are the unstable portions of the system's source code. For their identification, the system's history is mined for frequent modified file-pairs. Appropriate filtering methodology allows the selection and grouping of files with high relevance for the systems core or platform architecture.

One result is the evaluation of structural stability, i.e., how much of the structural dependencies have been added and to removed again from the system. This longitudinal view into the system informs about past re-structuring or re-engineering events, increasing or decreasing structural dependencies, stagnation etc. In contrast to metrics-based approaches trying, e.g., histograms depicting lines of code or Polymetric views, the detected structural dependencies provide qualitative feedback about the *actual* relations between entities such as features, modules, or files.

The second outcome is the detection of evolutionary anti-patterns in a system's longitudinal development. Anti-patterns are for instance short-lived source code modifications also called *day-fly*, recurring structural dependencies, or changing responsibilities. Since these anti-patterns are very likely undesired results of weak designed systems, their occurrence is an indication for possible design erosion or instabilities. Another interpretation is that the inappropriate knowledge about the correct usage of software components leads to incorrect solutions.

1.4.2.4 Alternate data sources

In contrast to the type of data exploited in our case study, it also possible to mine navigation information gathered during editing sessions [122], presumed data in sufficient quantity are available. The recorded data are for instance files which were inspected and modified for the fulfillment of a certain task. This editing information can be considered as the logical dependencies between these files. It would replace the information obtained from version control systems. The navigation information is usually finer-grained and with respect to the time dimension more actual than information gathered from version history. Recording editing sessions has more potential than pure recording of logical dependencies. For instance "copy, paste, modify" sequences could be recorded easily and would deliver interesting information how new code arises. Data obtained from editing sessions also have the advantage that files which are just opened to review some functionality but not altered or to read some documentation are captured as well. This type of dependencies are currently not stored in the source code repositories. A drawback is the requirement for special support by the tools used to open and modify the files.

1.4.2.5 Visualizations

Part of our approach are a number of visualizations for presenting the results in an appropriate form. These visualizations on various abstraction-levels—for instance on feature or code-level—support the assessment of the evolution. Other visualizations support the assessment of dependencies between modules and their respective files. Further interpretation is supported via visualizations of structural change patterns on method- and variable-level.

A frequently used concept in software analysis is the aggregation of lower-level information onto higher-levels or the abstraction of concrete code-level information. To obtain a holistic view with respect to historical dependencies, code-level information is abstracted onto the feature-level and the associated dependencies are projected onto the module structure of the system. The presentation of these dependencies between features facilitates estimation of maintenance efforts to fix problems concerning a single feature or a set of features.

A further technology used in our approach are visual clustering tools. Though discrete clustering delivers valuable results on structural level as draft for further refinement, they have some shortcomings in pointing out evolutionary hot-spots, i.e., cluster elements which have also a strong coupling with elements from other clusters. Furthermore, when applying a discrete clustering algorithm, the results may heavily depend on some initial assumptions or the selection of the distance metrics. More information about discrete clustering can be found for instance in [138]. A further reason for relying on continuous clustering is that we use historical information which represents primarily the likeness between different entities rather than concrete structural dependencies. For instance, a certain module structure may be useful with respect to system decomposition and anticipating future requirements though it had some shortcomings in the past. To avoid this situation we basically use visual clustering technologies to produce a layout for the system engineer where related structural entities are placed close together and unrelated entities are placed as fare as possible apart form each other. This map with the major related entities and their dependencies points out the hot-spots of a system from the evolutionary perspective. Affected parts of the architectural blueprint have to be validated against these designated hot-spots.

1.4.2.6 Continuous monitoring

A further interesting application of our approach is the continuous monitoring for the assessment of ongoing projects. Periodically or based on triggers generated from the version control system, the software system can be inspected for critical evolutionary dependencies or critical evolutionary patterns such as recurring changes of structural entities. On receiving the notification about critical patterns, the system engineers can react accordingly upon to fix possible design shortcomings in early stages of the development. For traditional development models such as the waterfall model with a well designed architecture, this information is less important since shortcomings in the architecture are difficult to correct. But the advent of extreme programming methods has created the demand for appropriate guidance of the development process. Incremental updates of the *Release History Database* are appropriate means to limit and control

the information flow into the monitoring system. Additionally, the application of a *temporal lens* puts the focus on the most recent changes or changes within any other timer-interval.

1.4.2.7 Application realization / framework plug-in

Analysis tools such as our proposed *EvoGraph* approach are instantiated within our *EvoZilla* framework and operate on a common databases. Other framework plug-ins may make direct use of the large amount of historical information already available. Advantageous is also the existing set of data evaluation and access functions allowing the fast implementation of new analysis approaches. This facilitates the development of new and the improvement of existing tools since common tasks such as data extraction and preparation already exist and have not be to reimplemented. Another benefit is the connection with the structural analysis processes via the *Release History Database*. Both, the structural and the evolutionary analyses benefit from accessing the same relational database. Thus, analysis results based on source code entities such as features, modules, file or method-level can be directly accessed by other approaches analyzing the same system.

1.4.3 Results

In the course of this thesis we developed a number of intermediate results: directly usable for other researchers and software maintainers are for instance the described approaches, the concept of building the *Release History Database*, or the program code. Not of direct use for others are the results from the case study. The may be of interest when other approaches also exploit the system used in our case study or comparative studies with other systems are required. Following we summarize the major results:

- *Release History Database*: we developed a relational database model to accommodate historical data extracted from the version control system, problem reports, or feature information. Moreover, import and filtering functions accompany the building process of this central data repository of our case study;
- to test our research hypothesis we used the *Mozilla Application Suite* which is a large scale Internet application. The source code represents more than five years of ongoing development and more than two million lines of C/C++ code. Other studies have used the *Mozilla Application Suite* as well which makes results even more interesting to compare and exchange.
- feature analysis: central to our analysis methods is the application of software features as the selection and observation unit. Features are also used in the *EvoGraph* approach to determine a core set of files for a detailed evaluation of structural dependencies. This is part of our systematic approach to identify entities with structural shortcomings reflected in a system's history;
- the analysis framework called *EvoZilla* is our testbed for approaches such as *EvoGraph*, *EvoTrace*, or *EvoFamily*. Other approaches may benefit from the reuse of existing functions;
- in the *EvoTrace* approach we examine the applicability of runtime data for evolutionary analyses. The dynamic information gathered from different versions of the instrumented application are imported into the *Release History Database* and exploited to analyze the evolution of the application;
- *EvoFamily* examines our approach for building a *Release History Database* from different products of a product family. Part of the *EvoGraph* approach is used to exploit the different product data. The commonalities are extracted and projected onto the system's module structure to reveal structural similarities between different products.

The main outcome of our research is the *EvoGraph* approach. Given as input the system's history and source code changes, the approach produces a qualified subset of entities with their corresponding dependencies. They represent the evolutionary hot-spots revealed from the pairwise changes, implicitly recorded in the change history. They are pointed out in visual form on behalf of the system engineer who has to interpret the results with respect to the as-designed architecture. The information can be conceived as the *evolutionary echo* of the as-implemented architecture. Dependencies can be interpreted as the architectural

linkage between source code entities or in the case of surprising results as the systems structural deterioration. The task of the system engineer is to validate these dependencies on the basis of the architectural blueprint and to classify them as harmful if necessary. Other key features and interesting properties with respect to our *EvoGraph* approach are:

- history & structure: we exploit both information spaces, first to select and point out entities with structural shortcomings (*evolutionary echo*), and second in the subsequent structural analysis step to identify structural dependencies;
- cross language analysis: the approach of extracting *string sequences* as identifiers from the respective source code enables the fact extraction from different types of source code and the subsequent analysis across language boundaries;
- performance: our approach is 10 to 100 times faster compared to traditional approaches analyzing structure and evolution. Reason is that we only need incomplete source code information obtained from source code changes instead of recreating a complete source code model;
- longitudinal view: intermediate result and input to further analyses is the reconstruction of the structural evolution of selected entities. Detailed information about new, updated, or removed structural dependencies build the input to pattern detection and stability assessment. Moreover, this longitudinal "elevation" of selected entities also enables the validation of success or failure of re-structuring or re-engineering events;
- evolutionary patterns: prerequisite for their detection is the fine granular time-interval for the analysis of source code changes. In the *EvoGraph* approach we track *every* change transaction of the source code—which can be up to several hundred for certain source files; and
- structural stability: our approach facilitates the assessment of structural stability on quantitative- (e.g., number of modified interfaces) and qualitative-level (e.g., occurrence of evolutionary patterns) as well.

In the following chapters the *EvoGraph* approach and the results are explained in detail. An elaborate case study shows the applicability, usefulness and efficency of our approach.

1.5 Threats to validity

Most relevant for the application of our approach is the choice of a relevant case study with respect to interesting evolutionary properties such as large, complex, and long-lived software systems which have to accommodate new or changing requirements. This increases the possibility to find logical couplings in the recorded history and to find structural shortcomings in the source code. Source code, release history or bug tracking data are sensitive information about a project's development and evolution. Therefore, obtaining access to this data of ongoing or recent industrial projects is difficult and the publication of the results is only possible under restrictive conditions.

Many approaches have been shown to work with small-sized case studies but rarely multi-million line software systems with industrial standard development process. Consequently we use data from large Open Source projects such as *Mozilla Application Suite* (Internet browser domain) or BSD (a family of operating systems). Besides source code they provide release history and bug tracking data and secondary documentation such as design documents, discussion forums and support by the development community. Some architectural information is publicly available too. A further plus for the *Mozilla Application Suite* is the professional style of development by a few specialists (including quality control and patch reviewing).

Historical data are available for at least 5 years for the *Mozilla Application Suite* and 10 years for BSD respectively thus a sufficient amount of release history information is available for both systems. With respect to quality we noticed some shortcomings with the *Mozilla Application Suite* system due to moved or identical entries. Since these anomalies are rather seldom they are correctable on an individual basis.

Sources of information about software features are system documentation, the executable program built from source code, or source code changes, i.e. source code deltas, and release documents. Detailed system

documentation may be inaccessible, inaccurate, outdated or not existent at all. The manual effort to map the abstract feature information from the documentation onto concrete pieces of source code entities is usually high and error prone. Extraction of software features from the executable program is a difficult and complex task. Especially applications with a graphical user interface require some kind of user interaction to activate a specific feature. Code deltas can be used to deduce from the description of features in the release document their respective location in source code. In either case the required manual effort is high and thus error prone.

Outdated source code can be difficult to parse due to language constructs such as old style C headers. Since we rely on source code deltas obtained from the version control system anyway, a customized source code parser seems to be an appropriate solution. This approach provides the most detailed information about a system's longitudinal behavior.

1.6 Résumé

As we will show, logical couplings are sufficient predictors to point out relevant structural shortcomings within a large, long-lived software system and provide clues about the structure of the as-implemented architecture. Our case study with the *Mozilla Application Suite* indicates the applicability and effectiveness in pointing out the major artifacts and their structural dependencies. A comparative architectural study we applied on the same software system was more time-consuming and encountered a number of problems with the used tools due to size of the system and therefore had to be limited to a subset of the overall system. With respect to small number of extracted facts from the architectural study, we identified the same entities with their strong structural coupling. In contrast to a pure architectural analysis approach, we were also able to provide detailed information about the longitudinal evolution, pointing out flow of structural dependencies between artifacts. Furthermore, we were able to identify *Bad Smells* [60] such as *God classes* in the structural dependencies of our case study. As the experiments with our prototype implementation have shown, the approach is fully automatable and therefore well-suited to monitor ongoing software projects. Though the system in the case study is Open Source Software the results are very promising with respect to the extracted information. Commercial projects with a higher standard in secondary documentation such as change- or problem reporting are even more promising to evaluate.

1.7 Architecture of this thesis

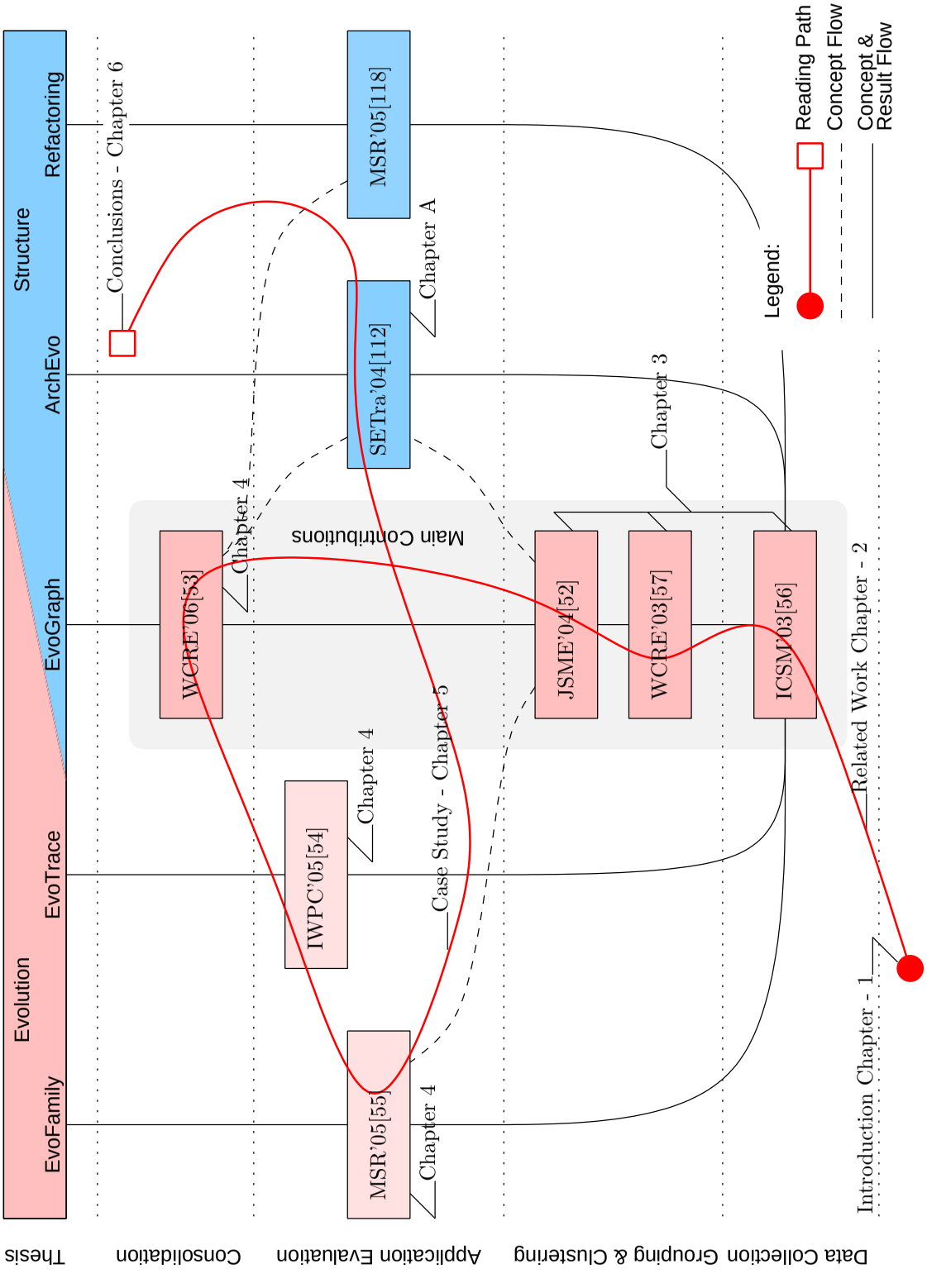
This section describes the architecture of this thesis and the logical relation of our selected publications. The citation numbers are according to the bibliography provided at the end of this thesis. Figure 1.6 indicates how the selected publications logically-relate to each other and contribute to this thesis. The main contributions can be summarized as follows:

- our first publication [56] presents the basic concept of a *Release History Database* which has been used for all subsequent publications;
- an evaluation with respect to historical dependencies and problem report information has bin done in [57];
- paper [52] is built logically on top of the previous publications and adds structural information to the evolutionary dependencies. Additionally, we have added more detailed feature information; and
- finally in [53] we describe how to obtain structurally meaningful information about the evolution of a software system.

Next, two publications are devoted to the evaluation of the possible other sources with respect to the detectability of dependencies and exploitability of dynamic information for software evolution analysis:

- an evaluation of the approach proposed in [52] with respect to product families has been presented in [55]. Keywords extracted from change log messages were used to find commonalities between the different products and to show their logical coupling with respect to module structure; and
- in [54] we evaluated the capabilites of dynamic information for software evolution analysis. The obtained results are linked on file-level with the data in the *Release History Database*. With respect to the *EvoGraph* approach, the analysis of runtime data provides more information about the actual implications of source code and configuration changes.

Figure 1.6 Mental map: Architecture and logical relation of publications which are exploited in this thesis.



To verify the architectural properties of our approach we have contributed the following publications which are also based on data and experiences gained from our work on the *Release History Database*:

- in [112] we joined the evolutionary and architectural information space to obtain a detailed view about the interplay between them;
- in [118] we have used the concept of the *Release History Database* to evaluate the evolution of a commercial software system. Based on the evolution data, we pointed out structural shortcomings and observed the success of re-structuring activities. This paper also supports the hypothesis that structure and its changes can be detected via their *traceable footprints*.

1.8 Further reading

Following the red line[1], further reading is organized as follows:

- An overview about related work with focus on software evolution analysis and reverse architecting is given in Chapter 2;
- In Chapter 3 we describe our approach for building a *Release History Database* for the systematic exploration of historical information obtained from software systems. We provide in depth information about the data extraction from the respective sources and the import of the prepared data. Subsequent processing steps such as the recovering of co-change transactions and grouping of logically-coupled entities are described as well. We also argue why we did not follow the classic approaches for partitioning software systems;
- Our approach for the systematic identification of relevant structural entities is presented in Chapter 4. We explain which resources can be exploited and how they can be used characterize structural dependencies and their effects on longitudinal development. Results are a systems hot-spots with notable influence on a systems evolution;
- The case study using the *Mozilla Application Suite* is presented in Chapter 5. We show how the structural shortcomings are reflected in the systems history. By means of a *stickiness view* we show how the logical coupling of the entities leads to clusters of structural dependent files. Detailed source change analysis informs about structural stability and reveals evolutionary patterns. We also demonstration how run-time and product family information can be exploited to characterize its evolution; and
- Chapter 6 concludes this thesis and indicates future work.

[1] In the city of Vienna the subway line U1 connects Kagran on the northern bank of the Blue Danube with Reumannplatz in the South.

Chapter 2

Related Work

While structural analyses such as the approaches of Müller and Klashinsky [107] or Biggerstaff [29] traditionally received a high degree of attention already for some decades, software evolution analysis appeared on the research agenda just recently. Hence, data format standards and analysis tools are still missing or under development. Following we review the most influencing and state-of-the-art literature with respect to software evolution analysis, architecture recovery, and software visualization.

2.1 Building a release history database

The first step in the application of the *EvoGraph* approach is the establishment of an appropriate data environment which we called *Release History Database*. Similar to our environment, Draheim and Pekacki proposed a framework for accessing and processing revision data via predefined queries and interfaces [43]. Linkage of their data model with other evolutionary project information—such as problem report data as required for our analyses—and making them accessible for external queries is beyond the scope of their work. Notable is also the approach called Kenyon by Bevan et al. [25] which is a data extraction, preprocessing, and storage backend designed to facilitate software evolution research. Though it was released as open source project [24] it does not seem to be actively maintained anymore. Newer proposals for instance concern the creation of a reference database. In [90] Kim et al. proposed an exchange language capable of making sharing and reusing software repository data as simple as possible. The proposal is part of the TA-RE project which aims at the building of a collection of release history data of different projects to facilitate mining activities and benchmarking.

Currently, some research work is done within the Controlling Software Evolution (COSE) [124] project to establish a meta model for release history data. A proposal for such a meta model has been documented in [102]. Another meta model for software evolution analysis has been proposed by Gîrba and Ducasse which is called Hismo [70]. History is modeled in their proposal as an explicit entity. It adds a time layer on top of structural information, and provides a common infrastructure for expressing and combining evolution analysis and structural analysis.

2.2 Release history mining

The analysis of software evolution based on release history information and the visualization of these results has been addressed by only a few researchers so far. Some of them also have used the *Mozilla Application Suite* for their investigations. For example, Mockus, Fielding, and Herbsleb used it in a case study about open source software projects [105]. They exploited data from CVS and the *BugZilla* problem tracking system but—in contrast to our work—focused on the overall team building 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.

With respect to release history mining, in [130] Taylor and Munro describe an approach based on revision data to visualize aspects of large software such as active areas, changes made, or sharing of workspace

between developers across a project by using animated revision towers and detail views. Since their approach is purely based on revision history, additional important information such as problem reports or feature data are not considered for visualization. Bieman et al. [28] used change log information of a small program to detect change-prone classes in object oriented software. The focus was on visualizing classes which experience common frequent changes, which they called *pair change coupling*. Instead of grouping logically-coupled objects they used standard UML diagrams together with a graph showing the number of pair change couplings between change-prone classes to visualize their analysis results. Similar to an earlier approach proposed by Gall et al. [62], Kemerer and Slaughter used modification reports as basis for their analysis [87]. They applied a refined classification scheme [126] on modification reports (corrective, adaptive, perfective enhancement, and new program) for an analysis of ordered change events and put quite some effort in the classification of change events. As a result, they were able to reveal different phases of a system's life cycle. Unfortunately, formal mechanisms to record such historical data during the development process are still not supported in software development tools. While they thoroughly investigated the longitudinal development of the system with respect to different phases, our focus is the visualization of—possibly hidden—dependencies between components of a system reflected by any kind of traceable pattern, e.g., commonly and frequently changed modules or common problem reports.

2.3 Evolution analysis

In [62, 63] Gall, Hajek, and Jazayeri examined the structure of a *Telecommunications Switching Software* (TSS) over more than 20 releases to identify logical coupling between system and subsystems; a similar study has been carried out by Bieman et al. in [28]. Based on release history data of this TSS, Riva et al. presented an approach to use color and 3D to visualize the evolution history of large software systems [64]. Colors were primarily used to highlight main incidents of the system's evolution and reveal unstable parts 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 work could benefit from the 3D visualization in that way that feature dependencies are visualized on a fine-grained temporal level.

Zimmermann, Diehl and Zeller presented a fine-grained analysis approach for CVS data that considered all kinds of entities starting from the statement-level [149]. Their ROSE prototype identifies common changes between syntactical entities rather than files or modules, which are the focus in our work. Hsi and Potts [80] 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). To obtain a holistic view about the evolution of a software system, the integration of data from code analyses and release history data is required such as they are provided by our *EvoGraph* approach.

In [92, 94] Lanza depicts several releases of a software system in a matrix view using rectangles. Width and height of the rectangles represent specific metrics (e.g. number of methods, number of instance variables of classes) according to the history of classes are visualized. Based on this generated 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). He analyzes the evolution of classes, whereas we focus on features and their couplings. However, a combination of both approaches could be promising.

2.4 Product family evolution

Within the EU projects ARES [10], ESAPS [14], CAFE [16], and Families [17] much work has been done in areas such as the identification of assets for product family architectures, evolution and testing of existing product families, or architectural models for product families. The project results have been compiled into a series of books: Jazayeri et al. [82] or van der Linden et al. [133, 134, 135, 136]. Closely related to our work on product family evolution is the approach presented by Riva and Del Rosso in [121].

They investigated the evolution of a family platform and describe approaches which enable assessment and reconstruction of architectures. In contrast to their work, we investigate the evolution of different variants to identify candidates for building a family platform.

2.5 Architecture reconstruction

Architecture reconstruction is concerned with the reconstruction of a system's architecture from the available artifacts such as source files. The reconstruction of the as-implemented architecture is a prerequisite for the evaluation of an architecture's stability.

Kazman and Carrière proposed with Dali [86] a workbench that aids a software engineer in extracting, manipulating, and interpreting architectural information through different views. By assisting in the reconstruction of architectures from extracted information, Dali helps to re-document architectures, discover the relationship between as-implemented and as-designed architectures, analyze architectural quality attributes and plan for architectural change.

Another view-based architecture reconstruction approach named NIMETA [120] has been proposed by Riva. He emphasizes the scrupulous selection of architectural concepts and architecturally significant views that are reflecting the stakeholder interests. Medvidovic and Jakobac proposed a bi-directional approach called Focus to evolve a system's architecture [103]. Their approach is driven by evolution requirements and applied iteratively. Each iteration is composed of two interrelated steps: architectural recovery and system evolution. Using this approach, the architecture of the original system is partially recovered, evolved to address new requirements, and enriched with detail in an incremental fashion. ArchView [111] proposed by Pinzger is an approach for the detection and visualization of bad smells in a system's architecture. He uses facts extracted from different releases of the source code and the release history of the system. Based on these facts metrics are evaluated which point out structural shortcomings. The evolution of the system on different abstraction-levels is depicted via the metric values using an extended version of Kiviat diagrams.

A relation to our work with architecture reconstruction exists insofar as we are interested in pointing out structural shortcomings hampering a system's evolution. Most reconstruction approaches are static and seldom take evolutionary properties into account. As a consequence, these shortcomings may remain undiscovered, harm architectural stability and aggravate maintenance. In [81] Jazayeri noted already that in opposition to traditional predictive approaches to architecture evaluation, retrospective analysis could be used for evaluating architectural stability by examining the amount of change applied in successive releases of a software product.

2.6 Coupling analysis

Though prior studies used co-changes for mining dependencies [27, 148], they basically relied on co-changes which represent *adhesion*. The less conspicuous co-changes which account for the *stickiness* have not been subject to detailed research work so far to characterize their properties on a global level and to draw their evolution over a systems life time.

With respect to coupling analysis, Zimmermann et al. inspected the release history data of several software systems for logical coupling between source code entities [148]. They drew the conclusion that augmentation of architectural data with evolutionary information could reveal new otherwise hidden dependencies between source code entities. With similar goals in mind, association analysis was also used in the work of Ying et al. [146] for change predictions. In contrast to their work, we systematically identify the evolutionary hot spots in a system and then add structural information from source code change analysis. However, our focus is not the prediction of candidates for possible source code changes.

2.7 Visualization of couplings

Our approach is related to visualization in that we filter logical coupling and use visual clustering techniques to provide a meaningful view on the *stickiness* between selected source code entities. With EvoLens [117] Ratzinger et al. proposed an approach which supports the projection of low-level couplings

into high-level views. One shortcoming is the limited filter capability which does not support the application of metrics for advanced filtering purposes. *SHriMP* is an earlier approach by Storey et al. [125] with focus on the presentation on software structure and code that combines both pan+zoom and fisheye-view visualization metaphors in a single view. As with *EvoLens* it is not designed to highlight dependencies which require expressive filtering mechanisms. In [36] Churcher et al. focus on coupling and cohesion of classes in a software system. The main difference is that we use logical coupling instead of structural properties. Beyer [27] used clustering techniques for grouping files based on Noack's [108] proposal for visual graph clustering. By adding a dynamic component Beyer improved his approach to study a system's evolution as animated sequences. His approach is called Evolution Storyboards [26]. Though based on the same clustering approach, the main difference of *EvoGraph* to Evolution Storyboards is that we go beyond the pure visualization of their evolution. Our focus is on the detection of structural shortcomings and revealing their evolutionary patterns.

2.8 Clustering of artifacts

Since our primary focus is not on developing a new clustering or modularization approach, we rely on existing technologies to group related artifacts. Wiggerts [138] provides a good overview about the possible application of clustering techniques with respect to re-modularization of legacy systems. Various clustering approaches and their aspects are discussed in detail. One conclusion is that a system will evolve further after it has been re-modularized and therefore the modularization will require incremental updates. For the study of evolving systems this seems a major drawback since trends will not be discernible from the clustered system and changes will appear suddenly in a discrete fashion. Dickinson et al. [42] used multidimensional scaling to cluster execution traces of faulty programs and compare the output with pre-determined test-cases. Our approach differs in that we operate on source-level and problem reports indicated on features and source-level elements. In a thorough study about the application of clustering techniques to software partition, recovery and restructuring Lung et al. [100] applied these techniques on different abstraction-levels of a system and different project stages. Besides a number of difficulties they encountered during the application of the discrete clustering approaches, they also concluded that no matter what clustering technique is adopted, there is always a chance that the method may generate unexpected results or will not generate expected results. Expert involvement is therefore recommended for postmortem analyses.

2.9 Dynamic analysis

Most related work we have seen so far, track the evolution of software systems by relying on static information about software artifacts or correlate the source code changes with respect to their programmers. For instance, Ball and Eick have proposed such approaches in [22, 45, 46]. Other reverse engineering approaches take the dynamic execution behavior into account and try to infer certain program characteristics based on these traces. In [74], for instance, Gschwind et al. presented an approach that allows one to identify how certain features within a program are implemented. This approach is based on execution traces and interactive program queries during the program's runtime. A similar approach is taken by the Smiley system presented in [71]. For this system, Goldman uses wrappers to log the interaction between an application and its external dynamic link libraries (DLL). This work facilitates the understanding of interactions between commercial off the shelf where no source code is available. Other research work on execution traces mainly focused on the visualization of execution traces [83] or detection of patterns to overcome the problem of size explosion [76]. Further, they have been used to dynamically discover likely program invariants that must be preserved when modifying and evolving source code [48].

Collberg et al. present an approach that takes possible executions into account by analyzing the evolution of the program's call-graph through static analysis [38]. This is accomplished by generating call-graphs for the different versions of the program, merging these call-graphs, and finally highlighting the differences between the call-graphs. Analyzing the differences in the call-graph, however, still falls short in getting a glimpse of the typical runtime behavior of the program to be analyzed since the call-graph does not give any information about how frequently certain functions are being invoked and hence does not provide a

deep insight into the communication patterns between different parts of the program.

2.10 Parsing and source code analysis

Due to the sometimes high frequency of source code changes in our case study and the small amount of applied deltas, the reconstruction and parsing of the complete source files is not necessary since we are interested only in changing dependencies and less interested in structural changes. Similar to island parsing [106, 127], we are searching for constructs which are required for our dependency analysis. Furthermore, frequently used words—such as common function names—are of minor interest and therefore sorted out. In [67] German studied the characteristics of source code modifications with focus on development process related issues. While we directly use the source code deltas to detect change dependencies, he uses two complete source code versions to extract the structural information. Moreover, our focus is on revealing the co-change triggering source code element.

In [101] Maletic and Collard proposed an approach to analyze source code differences based on its XML representation. The difference to our work is, that we are mainly interested to track the evolution of the change dependencies. Since they do not support structural dependency information, their representation would be of limited advantage for us. The approach proposed by Xing and Stroulia [143] is similar to Maletic's approach in the sense that they rely on a meta data representation of the software system. More recent is the work from Fluri et al. [59]. The focus of the proposed approach is on the identification of fine-grained changes within a set of frequently co-changed files. They have used Java files in an Eclipse environment which is quite homogeneous compared to the *Mozilla Application Suite* consisting of interface definitions, GUI definitions, glue code and other maintenance files.

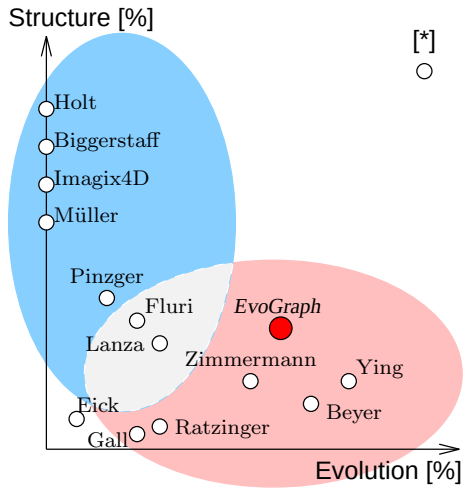
2.11 Résumé

Following we provide a not scientifically accurate (nsa) classification of selected approaches with respect to architecture and evolution. They are set into relation with a non-existent future approach denoted as [*], which finally fully integrates architectural and evolutionary analyses including prediction models into a single approach. Figure 2.1.(a) depicts the subjective result. The evolutionness (the degree a system supports evolutionary analyses) is aligned to the horizontal axis and indicated via pink colors while the structuredness (the degree a system supports structural analyses) of an approach is aligned to the vertical axis and also indicated via blue colors. Detailed results about percentage evolutionness and structuredness are listed in Figure 2.1.(b) together with the bibliography references.

Though there haven been many more approaches proposed, we can only cover a few of them as it is required to justify our approach. As outlined at the beginning of this chapter the structural analysis approaches do not consider evolutionary information. As a consequence, the earlier approaches are static approaches considering only a single snapshot in the life-time of a system. Holt, Biggerstaff, Müller or are typical approaches for structural analyses. Though evolutionary metrics can be incorporated, they are not explicit designed to support evolutionness. A full-fledged stat-of-the-arte commercial system for source code analyses is Imagix4D. Its focus is the provision of structural information. Evolutionary analyses are not supported.

An early attempt in the 1990s to reflect the notion of change in a retrospective analysis approach is Eick's proposal. His focus is the visualization of line oriented software statistics based on release history data. A few years later, Gall provided the first proposal to systematically exploit logical couplings. Besides files or modules no further structural information was used. The evolutionness is therefore higher compared to other approaches introduced so far. Based on Gall's findings about logical couplings, Pinzger proposed an analysis approach to detect structural shortcomings based on source code metrics and logical coupling. His proposal does not support the detection of evolutionary patterns. A forecast model is also missing. The evolutionness and structuredness is therefore moderate. Related with Pinzger's analysis approach is the proposal of Zimmermann. He systematically explored the relationship between structure and logical coupling of a software system. The results validate this assumption and also indicate a high variance between different systems. A step into the direction of a more detailed analysis of coupling between source

Figure 2.1 Mining classification with respect to properties of evolutionary and structural analyses.



(a) Mapping of approaches onto structural and evolutionary dimension

Approach	E[%]	S[%]	Year	Biblio.
Holt	0	90	1999	[79]
Biggerstaff	0	80	1989	[29]
Imagix4D	0	70	-	[3]
Müller	0	60	1988	[107]
Eick	10	10	1992	[45]
Pinzger	16	40	2005	[111]
Fluri	24	34	2005	[59]
Gall	24	4	1997	[63]
Lanza	30	28	2001	[92]
Ratzinger	30	6	2005	[118]
Zimmermann	54	18	2003	[148]
EvoGraph	62	32	2006	[53]
Beyer	70	12	2006	[26]
Ying	80	18	2004	[146]

(b) Detailed results for approach

code elements is the proposal by Fluri. He investigated the relationship between fine-grained source code changes and logical coupling.

Though Lanza uses several releases to extract metrics of a software system, he did not use release history or detailed structural information. Advantages is the possibility to detect evolutionary patterns and shortcomings in the visualized metrics. Thus we attest moderate evolutionness and structuredness. About at the same level of evolutionness as Lanza, Ratzinger proposed to use logical coupling to observe the success of re-factoring activities. The structuredness is low since the proposal is primarily designed the verify the success of changes and not to propose changes. Another approach with respect to visualization of evolutionary changes is Beyer's proposal which is related to partitioning as well. The evolutionness is high due to the used storyboard technique but has a low structuredness since the approach remains on file and module-level. Ying also relies on evolutionary information and provides a prediction model. Furthermore, structural information are used to identify interesting results obtained from their model.

In our *EvoGraph* proposal we use logical coupling to systematically point out structural shortcomings in a software system. Moreover, via fine-grained analysis of the release history we extract *evolutionary patterns in structural changes* which provides a high level of evolutionness combined with a moderate level structuredness.

Chapter 3

Building a Release History Database

The *Release History Database* is the central repository related to all activities concerning our retrospective software evolution analysis. It allows one a quick access to all types of information collected and computed for the artifacts under inspection. Modification- and problem reports imported from the respective sources of a software project constitute the basis for the evolutionary analyses. By adding related information such as product-, module-, or feature-information from secondary documentation or runtime data, the database turns into an important asset for structural analyses. Furthermore, the database lays the foundation for the creation of a holistic view on a system's evolution.

Large and complex software systems require the application of an efficient method to store and retrieve information. Our decision to use a relational database model to store the evolutionary information has inspired several other proposals for building storage models for evolution [43, 68, 70, 102]. The challenges in preparing such a database are the re-construction of change transactions, detection of anomalies and inconsistencies in a file's history, or reconstruction of a project time-scale from the available evidence. Once realized, the database is a valuable resource for software evolution analysis. In this chapter we will focus on the foundations of a software evolution analysis "data-warehouse". A detailed evaluation of the *Release History Database* is provided in in Chapter 5.

3.1 Information extraction from source code repositories

An important aspect in maintaining large software projects with world-wide contributors—such as open source projects—is that global changes affecting large parts of the repository are published in a transaction-oriented way. We have already defined a source code *change transaction* is a timely coherent sequence of check-ins of several, not necessarily logically-related files into the source code repository. When considering the logically-related files, we can empirically observe, that the process of applying the changes adheres—up to a certain extent—to the ACID paradigm known from database transactions. Overall objective of this process is to keep the repository in a well-defined state for all stakeholders. Since each transaction has to be prepared manually, it is more an organizational than a technical issue. Without enforcement of the ACID properties we would not be able to detect any logical coupling since then each file could be separately checked-in. The four ACID properties for *change transaction* can be characterized as follows:

- **A**tomicity: this property depends on the person who manually checks-in the source code changes and is therefore more difficult to maintain. Check-ins are usually not tested beforehand and therefore artifacts may not be checked-in;

- **C**onsistency: for organizational reasons, source code changes are implemented and tested outside the repository and then checked-in into the repository. The objective of the check-in operation is to transform the repository from one consistent state into another one. This can be expressed as a simple function transforming a source code revision into another one: $R' = f(R)$;

- **I**solation: files are usually locked by the version control system on behalf of the person which checks in the source code changes. Therefore no two changes can interfere with each other; and

- **D**urability: all source code changes are written back to the repository. The effect can only be undone via an explicit second change transaction;

The reason for the validity of the ACID assumption is the requirement to maintain the integrity (half implemented functions, syntax errors, undefined symbols, etc.) of such projects. The integrity itself is ensured via nightly builds and periodically applied regression tests. As a consequence, parameter type change, for instance, require the modification of both ends of an invocation relationship. Thus the callee and the caller have to be modified *at the same time* which emerges as *co-change* or in possibly ill-structured systems as *logical-coupling* in the course of their evolution.

Next, we focus on the extraction of the required information from the release history. Objective is the development of the required foundations to discover these logical-couplings.

3.1.1 Information extraction from CVS

Basically, CVS [35] is designed to handle revisions of textual information by storing deltas between subsequent revisions in the repository. This works well on text files such as source code since they can be compared line-by-line. Binary files can be stored in the repository as well, but they are not handled efficiently. CVS which is used as source code repository in our *Mozilla* case study has another major drawback: transactional information about co-changes are not recorded. Hence we have the reconstruct this information from the available evidence. First step is the exploration of the available data material.

3.1.1.1 Revision numbers

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

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

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

Branches are common to version control systems and indicate a “self-maintained” development-line [35]. Each branch is identified by its *branch number*. CVS creates a new branch number by picking the first unused even integer, starting with `2`, and appending it to the file’s revision number where the corresponding branch is forked off. For example the first branch created at revision `1.2` of a file receives the branch number `1.2.2` (internally represented as `1.2.0.2`). The main issue with branches is the detection of merges. Unfortunately this is not supported by CVS. With respect to our case study, an empirical evaluation indicated that most branches are for testing purposes such as a bug-fixes and therefore a further consideration is not required.

3.1.1.2 Version control data

In the repository the version control and historical data are stored in work-files. One work-file for each artifact. From there, log file information can be retrieved by issuing the `cvs log` command. The specification of additional parameters allow the retrieval of information about a particular file or a complete directory. 3.1 depicts an example log file taken from the *Mozilla* project showing version data of the source file *nsCSSFrameConstructor.cpp* as it is stored by CVS.

Listing 3.1 Sample of a CVS log file from the Mozilla project.

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

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

3.1.1.3 Sample CVS log information

Basically, a log file consists of several sections, each describing the version history of an artifact (i.e. file) of the source tree. The snippet of the CVS log in Listing 3.1 indicates the structure of information which can easily extracted via standard client programs. Sections are separated by a line of '=' characters.

For the population of our release history database we take into account the following attributes:

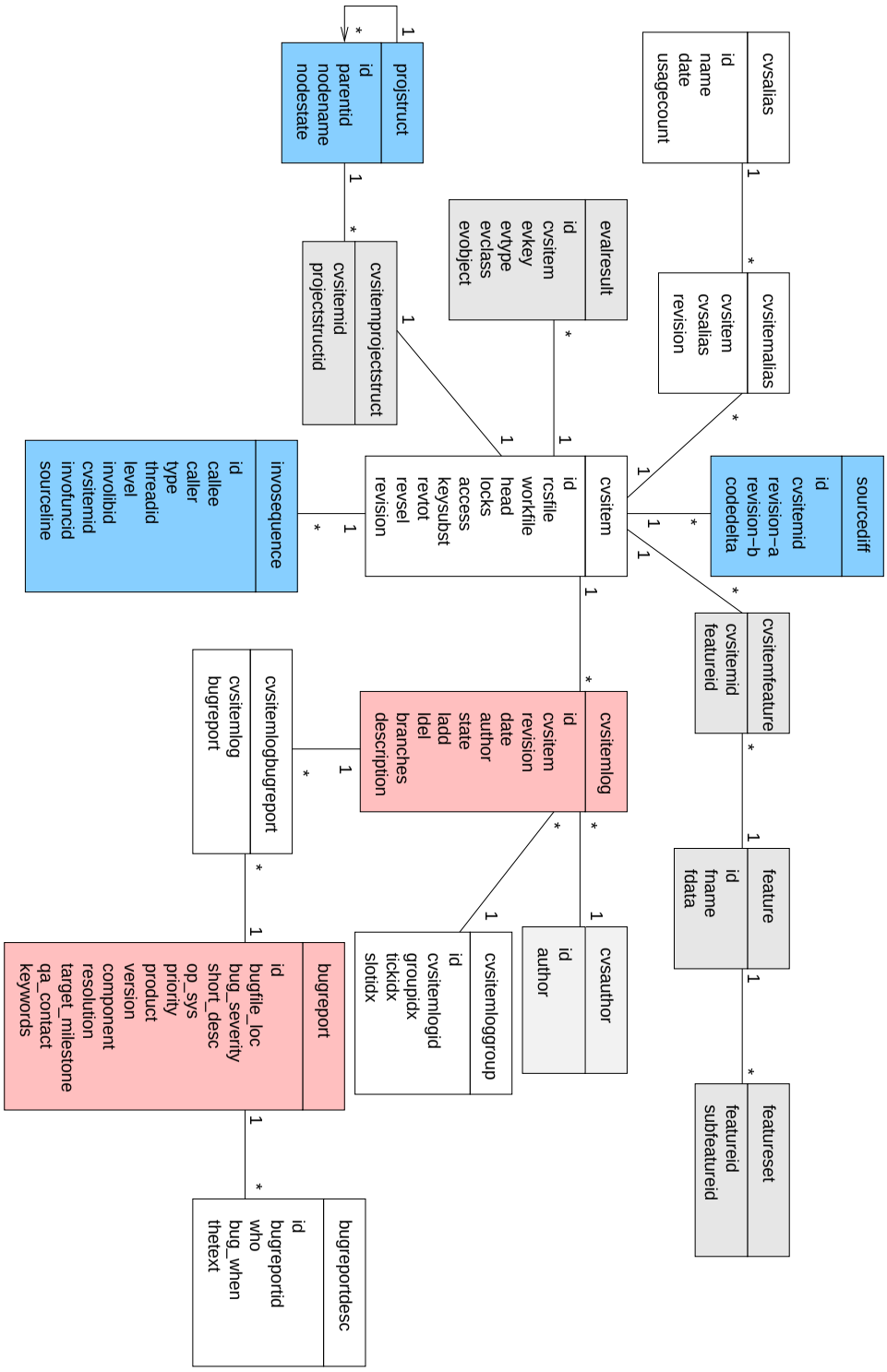
RCS file: the path information in this field identifies the artifact in the CVS repository;

symbolic names: lists the assignment of revision numbers to tag names. This assignment is individual for each artifact since revision numbers may differ;

description: lists the *modification reports* describing the change history of the artifact starting from its initial check in until the current release. Besides the modifications made in the main trunk all changes which happened in the branches are also recorded there. Reports (i.e. revisions) are separated by a number of '-' characters:

- the *revision* number identifies the source code revision (main trunk, branch) which has been modified;
- date and time of the check in are recorded in the *date* field;
- the *author* field identifies the person who did the check in;

Figure 3.1 Database scheme for the *Release History Database*.



- the value of the *state* field determines the state of the artifact and usually takes one of the following values: “Exp” means experimental and “dead” means that the file has been removed;
- the *lines* fields counts the lines added and/or deleted of the newly checked in revision compared with the previous version of a file;
- if the current revision is also a branch point, a list of branches derived from this revision is listed in the *branches* field (e.g. `1.638.4` in Listing 3.1);
- the following *free text* field contains informal data entered by the *author* during the check in process.

The above listed information is automatically extracted from the repository and imported in our *Release History Database* with the used schema as described in the next section.

3.1.1.4 A blueprint for release history information

Figure 3.1 depicts the most relevant entities of the *Release History Database*. Origin for all mining activities is the *cvsitem* table which provides the main information about the artifacts of the software system under inspection. Most other entities link to *cvsitem* via their foreign-key *cvsitemid*.

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

Problem reports (see Section 3.2) are directly imported from the problem tracking system into the *bugreport* table. The current attributes of this entity are derived from the *BugZilla* system and may be extended to address other problem tracking systems. Particularly, the link of problem reports with modification reports is important for software evolution analysis. We realized this link by the *cvsitemlogbugreport* table as *m:n* relation. The table contains the bug report numbers found in the modification reports together with the respective modification report ID.

3.1.2 Repository evolution

A snapshot of an example CVS repository is depicted in Figure 3.2. It shows a series of check-ins of two authors a and b, respectively. The different transactions are indicated via $T_{x,y}$ where x denotes the author and y the timestamp of the whole check-in transaction. Each check-in of a file is logged via a different time-stamp. Since only the currently affected file during a check-in is locked, different check-in transactions may run in parallel such as $T_{a,t_{50}}$ and $T_{b,t_{55}}$.

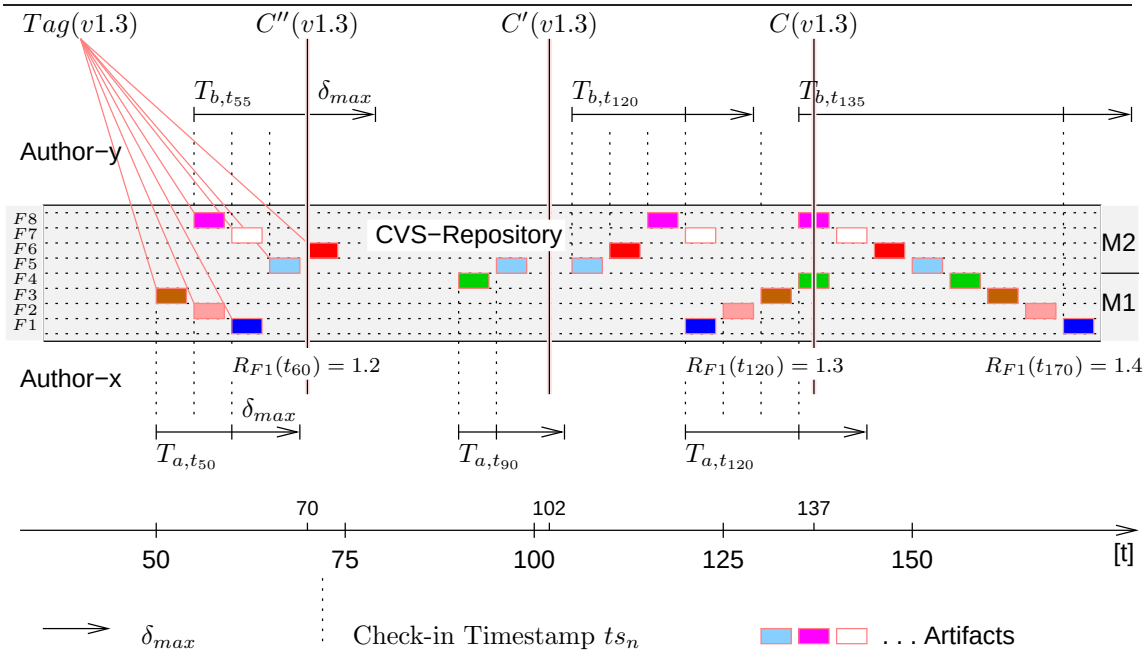
An important problem with respect to our case study were large source changes which affected hundreds or even thousands of files. Transaction $T_{b,t_{135}}$ symbolizes such an event. They have a wide spread effect on different modules and obfuscate the relevant co-changes. This data has to be filtered out or considered explicitly during further evaluations. Relevant for software evolution analysis are the cross cutting change transactions depicted as transaction $T_{a,t_{90}}$ which happen, as already outlined, less frequently compared to local co-changes.

Since CVS has no notion about co-change transactions, we need to recover this information from the available evidence. Notable two sources exist to recover this information:

- when logging is enabled the time-stamp of the commit—which is the same for all files of a change transaction—can be used. This information is recorded in a file called *CVSROOT/history*;
- another option are the time-stamps when an artifact is checked-in into the repository which is recorded individually for each artifact and is kept together with artifacts information in the repository. This data can be retrieved as log-information in plain text format.

Though the first source would provide more accurate data—its availability depends on the configuration of the repository—we have to use the latter one for the *Mozilla* case study to reconstruct the information.

Figure 3.2 Evolution of CVS repository with respect to different check-ins, recovery of co-change transactions and release date information



This implies that the transactions have to be recovered from the individual time-stamps of each checked-in file.

Characteristically for co-change transactions is their creation in an ad-hoc style. In contrast to database transactions, the check-in operations are not planned or tested beforehand. Possible threats to validity are:

- forgotten check-ins: changes in artifacts have been forgotten to check-in. As result the completing check-in operation has a large timely offset. Therefore, two co-change transactions are recovered;
- unrelated check-ins: the changes made in the software system have no logical relation but were contemporaneously checked-in with identical textual descriptions, e.g., one description for all changes. Consequently they are recovered as a single co-change transaction;
- different descriptions: two check-in operations may be logically-related but a different textual description was used for some reasons, e.g., typo or additional bug fix at one file. Two co-change transactions would be recovered; and
- large change transactions: changes which affect several hundred or even thousands of files require a large amount of time—sometimes even several hours. An approach with a too small maximum value for the time-window will recover several small co-change transactions instead of a single transaction.

Thus, it is not possible to reliably recover all “true” co-change transactions. To ease a further analysis of the recovered transactions, we use an extensible time-window with a “snap-radius” to capture check-ins of the same author. As it is simpler to split large transactions into sub-transactions than to recombine a set of small transactions, we do not consider the textual description at this stage as criterion in recovering a transaction. Possible sub-transactions are identified later via their differing or identical textual description and split off from the larger transaction. Thus, we use Algorithm 1 to detect the co-change transactions from the modification reports.

In the reconstruction of co-change transactions we use slightly different approach with respect to [111] to reflect the nature of manual check-in operations. A manual inspection of modification reports has shown that transactions are possible with a duration of more than one hour and also frequently modification reports with slightly different text occurred, e.g., forgotten files. As empirical results of our case studies have

Algorithm 1 Reconstruction of change transactions from sorted list of modification reports.

```
1: function GROUPLOGS
2:     δ_max = 900                                   ▷ default time-window is 900 seconds
3:     V ←'select * from cvsitemlog order by author,date'
4:     W ← {}                                        ▷ output list with transactions
5:     while V ≠ {} do                               ▷ while not empty
6:         R_i ← V.first()                           ▷ return first element and keep tail
7:         t_Ω ← R_i.time + δ_max
8:         T_x ← L_i                                 ▷ create new transaction record
9:         R_j ← V.first()
10:        while R_i.author = R_j.author ∧ R_j.time ≤ t_Ω do
11:            T_x ← T_x ∪ R_j                       ▷ add to transaction
12:            t_Ω ← R_j.time + δ_max                ▷ adjust time-window to snap radius
13:            R_j ← V.first()
14:        end while
15:        W ← W ∪ T_x                               ▷ add new transaction
16:    end while
17:    return W
18: end function
```

shown, the error rate is about the same for both variants. Since it is easier to separate them later if necessary instead of finding related groups, we build only a single group for those entries where the text differs but the ID of the author is identical. We therefore have the following different characteristics for recovering change transactions:

- the change log message is not considered since forgotten files in manual check-ins frequently have different messages;
- a dynamically extensible time-window of 900[s] captures also change transactions with more than one hour duration.

The recover transactions are stored in the relation *cvsitemloggroup* and can be uniquely identified via the assigned *groupidx*. Along with the group information we stored two other meta-informations:

- *time-slot* which determines the period between two subsequent releases and is derived from the determined release dates; and
- *tickidx* which is a fine-grained artificial time-scale base on a three hour interval. This time-scale is used to provide an alternative means of sampling for time-series analysis of the modification reports.

Later, we will use this recovered information to determine the logical coupling between files. Consequently, it determines the selection of entities for further analysis.

3.1.3 Release date synchronization - defining a global time-scale

A peculiarity of open source projects is the publication of the source code. It is provided in different flavors which also stem from different stages of the production process. In our analyses we use the source code changes and the modification reports provided directly by the respective repository. This information reflects the *continuous* stream of information recorded in the repository. It allows the reconstruction of any development stage at any individual point in time. In contrast to this, in structural analysis it is advantageous to use the released source code package instead of a reconstructed version from the repository. Consequently, to obtain a coherent view with other approaches the obtained timely information must be synchronized with the official release dates which have a *discrete* characteristics.

3.1.3.1 Software configuration management

Though we are not directly working on software configuration management [21, 39] we need to extend our vocabulary to ease the discussion about system releases and file revisions. We begin with the definition of

a *revision* as *an individual modification applied onto a particular file*. Consequently, a series of different, subsequent revisions determines the version of a file used in a specific configuration. A *revision-number* is therefore defined as *an ordinal number uniquely identifying a specific version of a file*. In configuration management the individual revision-numbers are not only impractical to handle but have also the drawback that for a selected configuration a large number of individual revision numbers have to be specified. To overcome this limitation the orthogonal concept of tag-names provide means to mark all individual revisions of a configuration. Thus a *tag-name* is defined as *a symbol assigned to a specific revision for each file of set*. During the data extraction from the source code repository, the tag-names are counted and stored in the table *cvsalias* of our *Release History Database*. The counter values are used to automatically identify the release date candidates.

Finally, we define a specific *software configuration* as *the set of files and their individual revisions which constitute a specific version of a software system*. The software configuration management system (e.g., CVS) keeps track of the individual configurations and provides facilities to extract a specific configuration from the system.

3.1.3.2 Release date recovery

The semi-automatic recovery of the release dates is based on the tag-names available in the repository. During the import process the appearances of the tag-names are counted and their time-stamps are recorded as well. Objective of the recovery process is the identification of possible release dates. The required heuristics to obtain usable results is the systematics of the tag-names and the release intervals (e.g., monthly, for the system in our *Mozilla* case study).

Algorithm 2 computes the "true" release dates $R''_{V_{i.j}}$ based on the beforehand mentioned input data. The results for the reconstruction of the release dates in our case study are listed in Appendix C.

One key-concept used in the approach is the assignment of different priority-levels to matching tag-names. For this purpose the *getpriority* function uses an ordered list of regular expressions $(r_0|r_1|\ldots|r_n)$, whereas the first expression has the highest priority. The priorities are then used as index to address the different FIFO-buffers of the priority queue. All relevant tag-name candidates for a single time interval are added to the priority queue from where the best candidate is chosen. Algorithm 3 outlines the applied strategy.

Usually the first entry in the priority queue is used as the result for the best matching release date in Algorithm 3. To consider possible important increases in the project size as well, we use an adaptable weighting scheme for the other entries of the priority queue. A 10% increase in the number of usage count of a tag-name is the default compensation for a lower priority. This has the effect that versions with sudden increases in the number of project files are considered as release versions (law of *Continuing Growth* [95]).

3.1.3.3 Synchronization

One problem with the reconstruction of information is the synchronization with information from external sources such as release dates. The release dates are required to obtain a consistent view with the development cycles of the system and also to synchronize with other analysis approaches such as *ArchView* which are usually applied on the released packages. Source code repositories record a "continuous" stream of source code changes. Though they are set to a "frozen" state to obtain a specific system release, the release date is usually different from the freeze date but the latter one determines the source code changes relevant for a given release. Consequently, release dates are arbitrarily selected dates which do not necessarily correlate with the time-stamps of the change transactions. Three different kinds of release dates have to be considered during evaluation in the *EvoZilla* framework. They are denoted as follows:

- $t(R_{V(i.j)})$ the official release date for release version $i.j$, e.g., as announced by the manufacturer. This information does not necessarily correlate with the time-stamps ts in the repository;
- $t(R'_{V_{i.j}})$ the date obtained from the officially distributed software packages, e.g., source code or binaries packages; and
- $t(R''_{V_{i.j}})$ the date obtained via revision information from the repository as approximation for the "true" release date.

Algorithm 2 Release date recovery from list of tag-names.

```
1:  function RECOVER
2:      A ←rhdb.sql("select * from cvsalias order by date, usagecount")
3:      P ← new priority_queue()                          ▷ allocate multiple fifo-buffers
4:      R'' ← {}                                          ▷ list of recovered release dates
5:      i_l ← 0                                           ▷ last interval
6:      n_l ← 0                                           ▷ last usage count
7:      while A ≠ {} do
8:          a_c ← A.first()
9:          p_c ← a_c.getpriority()                       ▷ priority is derived from predefined list of keywords
10:         i_c ← a_c.getinterval()                       ▷ based on selected release period, e.g., monthly
11:         n_c ← a_c.getusagecount()                     ▷ aliases were counted during import
12:         if i_c ≠ i_l then                             ▷ exploit the last interval
13:             R''_{V_{i_l}} ← CHOOSEBEST(p_c)           ▷ choose the best alias candidate for this interval
14:             P ← new priority_queue()
15:         end if
16:         if n_l < n_c then                             ▷ found new best usage count
17:             P.add(p_c, a_c)
18:             n_l ← n_c
19:         end if
20:         if n_l > n_c ∧ n_l < n_c 6/5 then              ▷ allow 20% deviation of best value
21:             P.add(p_c, a_c)
22:         end if
23:         if n_l = n_c then                             ▷ usage counts are equal
24:             P.add(p_c, a_c)
25:         end if
26:     end while
27:     return R''
28: end function
```

Algorithm 3 Choose best tag-name from priority queue.

```
1:  function CHOOSEBEST(P)                                ▷ P is our priority queue
2:      a_b ← null
3:      p_b ← 0                                           ▷ best priority
4:      n_b ← 0                                           ▷ best usage count
5:      while P ≠ {} do
6:          a_c ← P.first()                               ▷ get first alias from the priority queue
7:          p_c ← a_c.getpriority()                       ▷ current priority
8:          n_c ← a_c.getusagecount()                     ▷ current count
9:          if n_b + (p_c − p_b) n_b/10 < n_c then        ▷ plus some tolerance
10:             a_b ← a_c
11:             p_b ← p_c
12:             n_b ← n_c
13:         end if
14:     end while
15:     return a_b                                        ▷ return best rated alias from this queue
16: end function
```

Theoretically, the following relation should hold for every release of a software system:

$$t(R''_{V_{i.j}}) \leq t(R'_{V_{i.j}}) \leq t(R_{V_{i.j}}) \tag{3.1}$$

Threats to validity are unsynchronized computer clocks or manipulations of files before packaging. Another problem is the time difference between dates when the release snapshot (code freeze) is taken and the official release date. Thus, $T_{a,t_{90}} \in C(v1.3)$ as depicted in our example might be true (see Figure 3.2). $C(x)$ describes the source code configuration for a given time-stamp or release version. Consequently, we obtain three different configurations from the repository: $C(x)$ as the "official" one via the release date; $C'(x)$ via the file change date from the distributed packages; and $C''(x)$ via information from the repository.

To resolve this confusion, we evaluate the *tag* information [35] available in the repository of our case study. A tag-name is a symbol assigned to a specific revision to each file of a selected set. Thus it is possible to tag the revisions of all files which are used to build a specific release of a software system. In Figure 3.2 $Tag(v1.3)$ assigns a symbolic name to the most current revision of each file which complies with the release configuration $C(t_{85})$. We therefore exploit this symbolic information to obtain the desired approximation $t_(R''_{V(i.j)})$ for the "true" release date. Changes to the configuration after this date have to be considered for evaluation of the next release of the system.

3.2 Problem Reports

Problem reports aka *bug tracking data* are an important source to qualify and link related modification reports. The information available in the *Mozilla Application Suite* case study stem from the *BugZilla* problem tracking system. Linkage with the version control system is provided via means of problem report IDs added as free text to the description fields of the CVS modification reports. Consequently, this linkage as to be validated to obtain a trustworthy data basis.

Potential for evolutionary analyses is in the additional information of the problem reports which provide some description about the causes of changes. Furthermore, problem reports appearing in different co-change transactions may indicate continuous problems or inadequate solutions.

3.2.1 BugZilla

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

A sample of the available information is provided in Listing 3.2. Besides some administrative information such as contact information, mailing addresses, discussion, etc., the problem report also provides interesting information for the evolution analysis such as bug severity[1] of the affected product or component:

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

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

product: determines the product which is affected by a bug. Examples in the *Mozilla Application Suite* are Browser, MailNews, NSPR, Phoenix, Chimera;

component: determines which component is affected by a bug. Examples for components in the *Mozilla Application Suite* are Java, JavaScript, Networking, or Layout;

dependson: declares which other bugs have to be fixed first, before this bug can be fixed;

blocks: list of bugs which are blocked by this bug;

[1] *http://www.mozilla.org/quality/bugzilla-code-definitions.html*

Listing 3.2 Sample BugZilla data in XML format as extracted from the problem tracking system.

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

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

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

Aside from the descriptive information offered by the problem reports, we also exploit part of the reports as filter criterion in the determination of the logical coupling. A more restrictive filtering, for instance, could exclude all changes which relate to enhancements.

3.2.2 Plausibility check of problem report IDs in modification reports

As already stated above, the linkage between modification- and problem-report information has to be validated since the information was entered as free text. We validate the linkage via a set of regular expression obtained from a manual inspection of reports containing IDs in the range of valid problem reports but not matching regular expressions. In conjunction with the regular expressions, we use the following meta information to describe related actions:

- *confidence* describes the reliability of the result. A sequence "bugid=100069" is rated as high, whereas a sequence looking like "32767" is not considered as problem report ID. For the confidence we use a simple three-level schema (low, medium, high);
- *positive match*: if the expression matches, it supports the hypothesis that the number found was a problem report ID;
- *negative match*: indicates that the number is not a problem report ID and used in a different context, e.g., used as identification for a patch or an attachment;
- *complex test*: the declared regular expression is used as prefix a general expression to detect one problem report ID or a list of IDs. The matching process of the whole expression restarts if a new sentence indicated by one of the following four characters `.:;)` is detected; and

- *simple test*: in contrast to the above, the regular expression is matched as declared.

The results of this plausibility check is used to update the m:n relation between *cvsitemlog* and *bugreport* with the confidence value. Apart from uncovering this linkage, the import process itself is simple and straight forward due to the data provided in XML format.

3.3 Features

3.3.1 Feature extraction

The goal of our feature extraction process is to gain 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 our *Release History Database* with observations about particular releases of a product and can be used to apply evolutionary analyses on the feature-level. Results of the analyses support the illustration of dependencies and changes in large software systems. We restricted ourselves on dynamic feature analysis, since we are basically interested on a representative set of test-data for the case study with the *EvoGraph* approach.

The used extraction process is based on *Software Reconnaissance* analysis technique [139, 140] and utilizes code instrumentation for its application. This approach using GNU tools [8] has been also addressed in brief by Eisenbarth [47]. Concept analysis is used to validate our assumptions about the composition of features within the application. Currently, the whole feature extraction process is limited to file-level analysis but could be extended onto method-level as well.

3.3.1.1 Extraction process

The extraction process as depicted in Figure 3.3 resembles a simple pipe and filter architecture and is therefore well-suited to be operated in traditional batch style. This is useful, since some of the following steps are computationally intensive. Therefore, we first create all necessary data and then initiate the analysis process. The extraction process is as follows:

1. the application executes on of the defined scenarios (see also Table 3.1). 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 [72] 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 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 *Release History Database* and replaced by their corresponding *cvsitemlog* IDs. Output of this step are the binary relationships between file IDs and scenarios IDs;
4. from the binary relationship data a single concept lattice [97] is generated (see Figure 3.4(b));
5. the analysis of the concept lattice identifies the computational units (i.e., in our case files) specifically required for a feature. Hence it provides the mapping of features onto files. The result of this step is a list of file IDs required to realize a specific feature; and finally,
6. a Perl script imports the extracted feature information into the *Release History Database*. This feature information extracted from a specific release is then used as approximation about the implementation of all other releases.

The critical part in the feature extraction process is the identification of the computational units. Following we give an overview about the theoretical foundations and provide a small example with relation to our *Mozilla* case study.

Figure 3.3 Feature extraction process from runtime information.

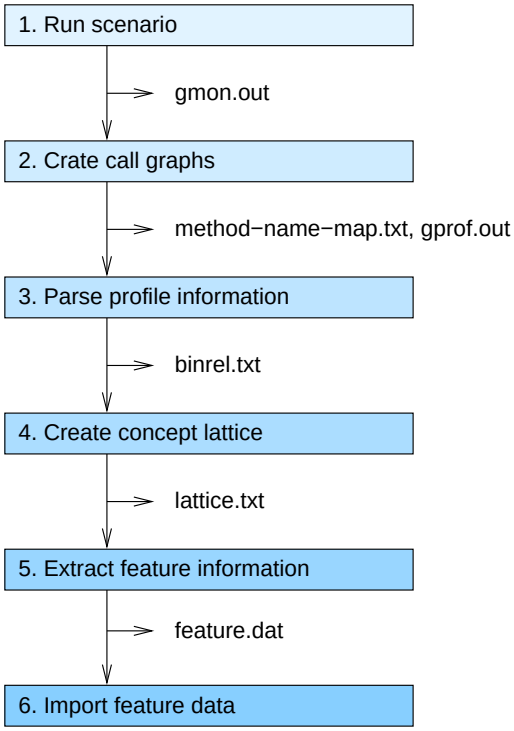


Table 3.1 Scenario definitions for runtime data generation.

Scenario	Description
sNull	Mozilla start / blank window / stop
sTC-HTTP	TrustCenter.de via HTTP *http://www.trustcenter.de/*
sTC-HTTPS	TrusterCenter.de via SSL/HTTP *https://www.trustcenter.de/*
sTC-File	read TrustCenter.de from file
sMathML	mathematics in Web pages
	http://www.w3.org/Math/testsuite/testsuite/General/Math/math3.xml
sAbout	"about:" protocol

3.3.1.2 Formal concept analysis

Formal concept analysis is a mathematical technique that provides insights into binary relations. The mathematical foundation of concept analysis was laid by Birkhoff [30] in the 1940s. Primarily Lindig and Snelting have introduced concept analysis to software engineering [99]. With respect to feature extraction [47] it has shown to be well-suited to point out dependencies of the data in the feature extraction process by providing a better insights into their binary relations. The mathematical details however can be found for instance in Ganter and Wille [66]. A good overview about *formal concept analysis* with respect to information science has been published recently by Priss [115]. Readers which are more interested in *formal concept analysis* are encouraged to use one of the referred resources.

3.3.1.3 Feature extraction and formal concept analysis

Concept analysis uses the terms objects and attributes which we map onto files (functions, methods) and scenarios (scenario in terms of a use case scenario). Consequently, we provide a basic scenario to extract some *Mozilla* features (see also Chapter 5). Table 3.1 lists details. Here, we use 6 attributes (scenarios)

Table 3.2 Concept matrix for the defined scenarios and expected file-sets.

\ Attribute → / ↓ Object \		0 sNull	1 sTC-HTTP	2 sTC-HTTPS	3 sTC-File	4 sMathML	5 sAbout
0	oCore	x	x	x	x	x	x
1	oHtml		x	x	x	x	x
2	oFile				x		
3	oNetwork		x	x		x	
4	oHttp		x	x	x	x	
5	oHtml			x			
6	oMathML					x	
7	oBlank	x					
8	oAbout						x

to obtain 9 objects (file-sets) as listed in Table 3.2. It suggests already that no trivial $1:1$ mapping of scenarios onto file-sets for all features exists, since many of the features are composed of other smaller features. Hence, the appear in different scenarios. For instance feature fHttp is used in the scenarios sTC-HTTPS or sMathML and consequently the information obtained from both execution scenarios has to be considered in the extraction process.

To obtain a better insight into the composition of features, we generated a concept lattice from the data in the concept matrix. Figure 3.4(a) depicts the result. By convention, on top of the graph is the most general Concept (set of programs which implement all features required by the different scenarios) and is denoted by $\top$. It has all formal objects in its extension. Its intension is often empty but does not need to be empty. The "smallest" Concept resides on the bottom (set of programs which are common to all scenarios = core functions) and is usually denoted by $\bot$. In our graph denotes as Core. It has all formal attributes in its intension. If any of the attributes exclude each other, then the extension of the bottom Concept must be empty (which should never be the case in a software system). The nodes in Figure 3.4(a) represent the formal concepts. Labels of the formal objects are noted on the left side below and formal attributes above the respective nodes. Open circles indicate empty Concepts (Concepts without own attributes), whereas filled circles indicate the Concepts with own attributes.

To obtain the file-set for a certain feature, we need to compute the difference between the extension of the respective Concepts. As examples we use the features fHttps and fHttp. The lattice depicted in Figure 3.4(a) suggest that feature fHttps can be directly derived from the objects of the Concepts 1 and 2 which have the scenarios {sTC-HTTPS} and {sTC-HTTPS, sTC-HTTP}, respectively, as their intension. Based on the file-sets obtained from the runtime information during the execution of the scenarios, the extraction of feature fHttps can be nicely written as the following set operation:

$$\text{fHttps} = (\text{sTC-HTTPS}) \backslash (\text{sTC-HTTP}). \tag{3.2}$$

This is possible since TC-HTTP represents basically the only sub-Concept of TC-HTTPS. The sub-Concept-super-Concept relation is depicted in Figure 3.5. In our second example the situation slightly different, since feature fHttp is used in three different scenarios: sTC-HTTPS, sTC-HTTP, and sMathML. To obtain the desired file-set we subtract the extensions of Concept 3 from Concept 2:

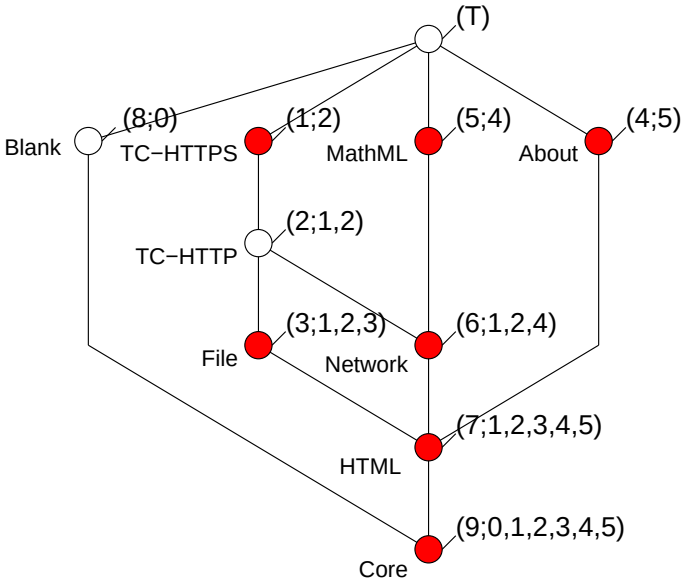
$$\text{fHttp} = (\text{sTC-HTTP}) \backslash (\text{sTC-File}). \tag{3.3}$$

Or alternatively, we can build the intersection of the extensions of Concept 2 and 5, and then we subtract the extension of Concept 4:

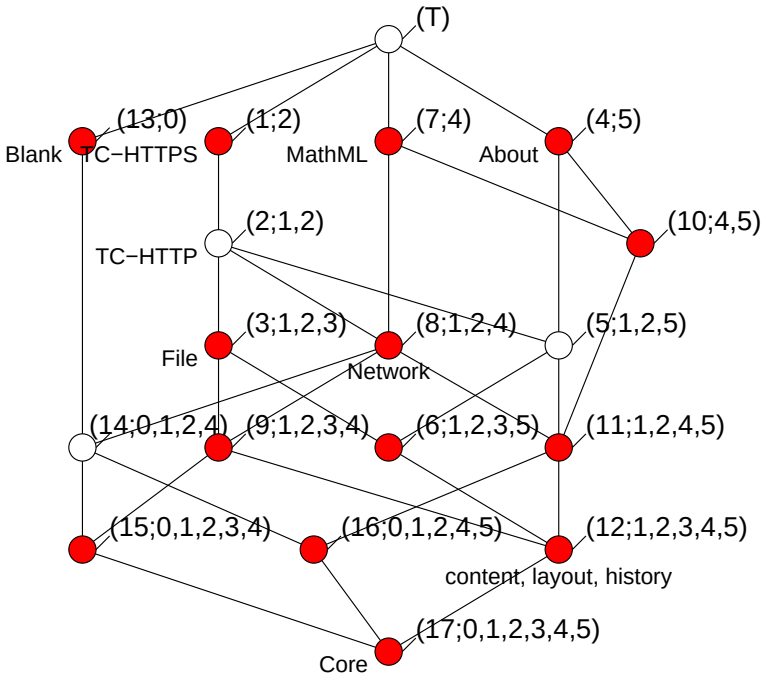
$$\text{fHttp} = (\text{sTC-HTTP} \cap \text{sMathML}) \backslash (\text{sAbout}). \tag{3.4}$$

Evaluating the actual runtime information provides a more complex picture about the composition of features from the source code. Figure 3.4(b) depicts the actual concept lattice derived from the runtime data. Though the upper part of the concept lattice is similar to the expected results, unexpected and new Concepts appeared. They increasing the number of Concepts from 9 to 17. Concept 15 is for instance

Figure 3.4 Concept lattice deduced from concept matrix and the actual results obtained from the *Mozilla Application Suite* as Hasse diagrams.

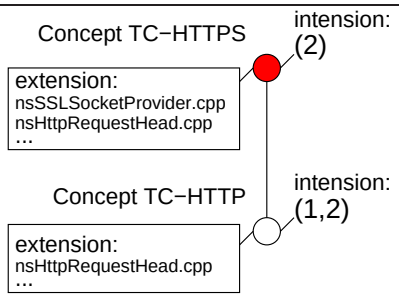


(a) Concept lattice deduced.



(b) Actual concept lattice.

Figure 3.5 Example for sub-Concept-super-Concept relation.



is such an unexpected Concept since during the execution of the scenario sNull, some network functions were used as well. Another surprise is Concept 13 which adds a single file to its extension. The file *xpfe/components/timebomb/nsTimeBomb.cpp* is used only in this scenario.

Though there are a number of changes in the concept lattice, the extraction of the fHttps feature remained equal since there is only one path down from Concept 1 (i.e., Concept TC-HTTPS) to 2 (i.e., Concept TC-HTTP). Hence, we need collect the formal objects only for this path. For feature fHttp the situation is different, since we have several paths to travel down from Concept 2 (i.e., Concept TC-HTTP) to the bottom Concept. Now, formula 3.3 is not sufficient anymore since a common sub-Concept does not exist anymore. Instead of Concept 7 as in lattice 3.4(a) we have now 4 different sub-Concepts (14, 9, 6, and 11) which are connected via different paths. As a consequence, we have to adapt Formula 3.4 to reflect this situation. We therefore add scenarios which have a common sub-Concept with Concept 2 on one of the paths down in the concept lattice. Formula 3.5 describes the final set-operation:

$$\mathrm{fHttp} = (\mathrm{sTC\text{-}HTTP} \cap \mathrm{sMathML}) \backslash (\mathrm{sNull} \cup \mathrm{sTC\text{-}File} \cup \mathrm{sAbout}). \tag{3.5}$$

Based on the runtime data, we can compute the file-set for feature fHttps with Formula 3.2. This results in six files for release *0.9.2* of the *Mozilla Application Suite* as listed in Listing 3.3. The same

Listing 3.3 Resulting files for feature fHttps of *Mozilla Application Suite* release *0.9.2*.

```
security /manager/boot/src/nsSecurityWarningDialogs.cpp
security /manager/ssl/src/nsNSSCallbacks.cpp
security /manager/ssl/src/nsNSSCertificate.cpp
security /manager/ssl/src/nsNSSIOLayer.cpp
security /manager/ssl/src/nsSSLSocketProvider.cpp
netwerk/socket/base/nsSOCKSSocketProvider.cpp
```

query executed on the runtime data for release *1.3a* yields two additional files (see Listing 3.4). A manual inspection revealed, that these files existed already in *0.9.2* and were commonly used during the execution of *all* scenarios.

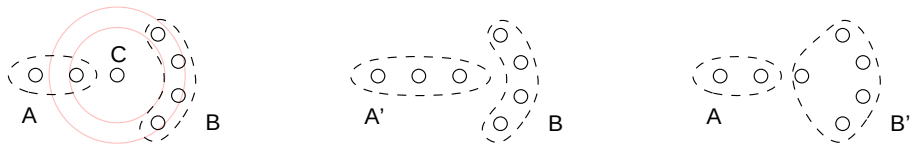
Listing 3.4 Additional files for feature fHttps in release *1.3a*

```
security /manager/ssl/src/nsNSSComponent.cpp
security /manager/ssl/src/nsNSSModule.cpp
```

3.3.1.4 Further use of concept analysis

The derived concept lattice depicts already the difficulties which have to be expected using more complex scenarios for feature extraction. A single concept lattice of large sets of objects and attributes can become

Figure 3.6 Example for different results based on the selected metrics [138].



(a) initial situation (b) single linkage (c) complete linkage

fairly large and complex. Visualizations of concept lattices are only of interest if they are not too messy to be comprehensible for a human user. The only information which such visualizations provide is that the underlying lattice is complex. Lindig and Snelting [98] use this as an advantage in one case: by showing that the concept lattice of dependencies between different pieces of software code is extremely messy, they provide an argument for not attempting to re-engineer such code. Experiments with the *Mozilla Application Suite* used in our case study indicated the complex structural relationships within the system. From the defined twelve scenarios in our case study we obtained more than 90 elements for the concept lattice which rendered it difficult to obtain useful file-sets from the lower Concepts. Consequently, for the extraction of feature information we will rely on an approximation which uses the unique parts of the Concepts.

3.4 Grouping of related release history information

Prior to further analyses, the large amount of dependencies obtained from the release history needs to be arranged in a useful way to facilitate the further discussion about shortcomings in the system. As already pointed out, logical couplings are not randomly and equally distributed over the whole structure of a system, rather they reflect the frequent co-changes of dependent structural entities.

In re-engineering, clustering approaches are frequently used to improve a systems module structure via re-modularization based on structural coupling metrics. The goal is to produce an improved version with respect to the decomposition of the system. An often mentioned problem with clustering methods is, that many methods impose a structure rather than finding "natural" clusters. A solution would be to use appropriate metrics and algorithms to impose the "correct" structure [138]. Decomposing an ill-structured system to obtain an initial proposal is a feasible and useful method. However, it can be assumed that an "appropriate" algorithm is not known a-priori with respect to the concealed evolutionary and structural shortcomings within large and complex software systems.

3.4.1 Hierarchical clustering

One frequently applied method to cluster entities are agglomerative hierarchical algorithms. Update rules—they are used when the similarities can not be computed from the original data—for the similarity measures after joining or splitting clusters are an example for the impact on the overall result. They play an important role, for instance, in how the different small clusters should be combined into larger clusters. When two clusters C_x and C_y are joined, the update rule computes the similarity between an already existing cluster C_z and this new cluster. The following *single linkage* rule yields that cluster B is as similar to $A \cup C$ as it is to the most similar of A or C:

$$singlelink(B, A \cup C) = max(sim(B, A), sim(B, C)).$$

In contrast to the previous rule, the *complete linkage* returns as result the least similar of the old clusters which would be B:

$$compllink(A, B \cup C) = min(sim(A, B), sim(A, C)).$$

Assuming the situation of Figure 3.6(a) with the *single link* update rule, the results are usually non-compact and isolated clusters (see Figure 3.6(b)). In contrast to this, the *complete linkage* update rule will find

compact cluster which are not very well separated (see Figure 3.6(c)). The example also illustrates that a single entity can have a great influence on the resulting clustering.

Consequently, instead of proposing a new decomposition of an existing software system based on the logical couplings, we use this information to point out the structural shortcomings via visual clustering. The advantage is that critical constellations are preserved and just pointed out. To draw a conclusion and react accordingly such as applying an appropriate design pattern lies in the further responsibility of the system engineer.

3.4.2 Introduction to multidimensional scaling

The goal of multidimensional scaling (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:

$$\mathrm{Stress}_D(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \left(\sum_{i \neq j} (D_{i,j} - \|\mathbf{x}_i - \mathbf{x}_j\|)^2 \right)^{\frac{1}{2}} \tag{3.6}$$

where the outer square root is just a convenience that gives greater spread to small values [33].

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 [33].

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 artifacts a_n which refer to a particular problem report via their modification reports m_k (linkage modification report– problem report; see also Figure 3.8):

$$d_i = \{a_n | a_n \mathsf{R} m_k \wedge m_k \mathsf{R} p_i\}. \tag{3.7}$$

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*.

$$\mathrm{dist}(d_i, d_j) = \begin{cases} 1 & \text{if } p_i \not\mathsf{R} p_j, \\ \frac{1}{2}(1 - \frac{n}{\min(s_i, s_j)}) & \text{if } p_i \mathsf{R} p_j \end{cases} \tag{3.8}$$

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 weak coupled 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.

An alternative way of specifying distances is to use edges and weights. We use a logarithmic function to emphasize the connections with higher values, while connections with lower values are weakened. This has the effect that the nodes with stronger connections are moved closer to each other than nodes with only a few connections. The weight for an edge between two nodes v_i and v_j are computed by the following formula, whereas n specifies the current number of connection between the two nodes and n_{max} the maximum number of connections between two nodes of this graph:

$$\mathrm{weight}(v_i, v_j) = 10 - \lfloor \frac{\ln(n)}{\ln(n_{max})} * 8 + 1.5 \rfloor \tag{3.9}$$

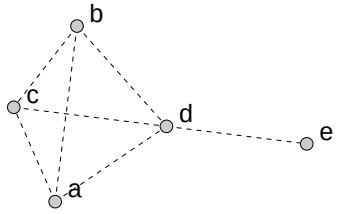
All weights are mapped by the above formula onto a range of $[1..9]$ where 9 means the closest distance. Other integer values not covered by the given range cause the visualization program *Xgvis* [33] to hang or crash. Now we just need to define when two problem reports are linked: p_i and p_j in the *Release History Database* are linked via a software artifact a_n if a modification report m_k exists such that

$$a_n \mathsf{R} m_k \wedge m_k \mathsf{R} p_i \wedge m_k \mathsf{R} p_j \tag{3.10}$$

Figure 3.7 Dissimilarity similarity matrix $\mathfrak{D}_G$ and resulting graph G.

$$\mathfrak{D}_G = \begin{pmatrix} 0 & 1 & 1 & 1 & 2 \\ 1 & 0 & 1 & 1 & 2 \\ 1 & 1 & 0 & 1 & 2 \\ 1 & 1 & 1 & 0 & 1 \\ 2 & 2 & 2 & 1 & 0 \end{pmatrix}$$

(a) Dissimilarity matrix



(b) Resulting layout

or two modification reports m_k, m_l exist such that

$$a_n \mathsf{R} m_k \wedge m_k \mathsf{R} p_i \wedge a_n \mathsf{R} m_l \wedge m_l \mathsf{R} p_j. \tag{3.11}$$

3.4.2.1 Example

To illustrate the optimization process we use a simple graph consisting of 5 nodes with the following edges and weights: $G = \{(a, b, 1), (a, c, 1), (a, d, 1), (b, c, 1), (b, d, 1), (c, d, 1), (d, e, 1)\}$. *Xgvis* automatically transforms these 3-tuples into the *dissimilarity matrix* depicted in Figure 3.7(a) and the missing distance values are completed by *Xgvis*. The graph after optimization with *Xgvis* is depicted in Figure 3.7(b) (settings and result: dim=2, metric, krsk/sh, $Stress = 0.1202$).

In our example, we use edges and weights to describe the similarity between nodes of the project structure. Formally, two nodes v_i and v_j of the graph G are connected if a path within the project tree exists between these nodes such that we can define a relationship R in the form $v_i \mathsf{R} v_j$, or alternatively the nodes share a problem report, denoted as $v_i \mathsf{R} p_n \wedge v_j \mathsf{R} p_n$. The weight for an edge between the nodes is computed via Formula (3.12), in which n specifies the current number of connections between the two nodes and n_{max} the (global) maximum number of connections between any two nodes of the graph.

$$\text{weight}(v_i, v_j) = (-1)(\frac{n}{n_{max}})^k + o \tag{3.12}$$

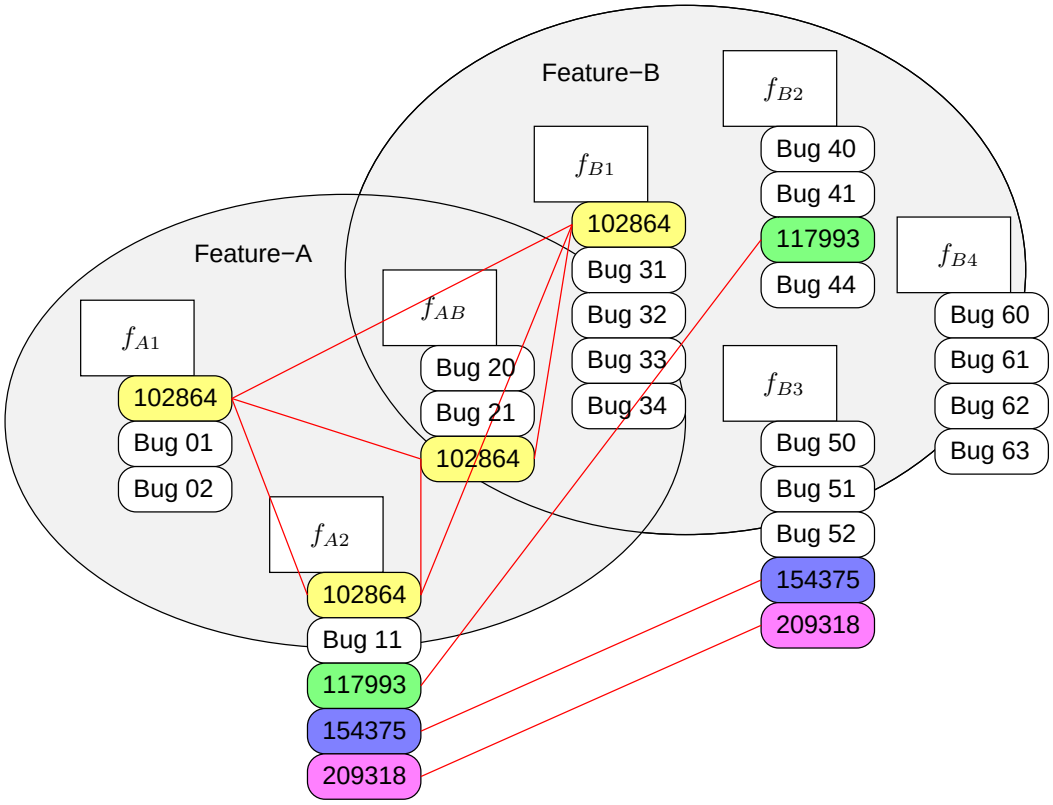
For an offset $o = 1$ all weights are mapped by the above formula onto a range of $[0..1]$ where 0 means the closest distance [33]. Since a weight of 0 is disadvantageous in the optimization process, because objects would be placed on the same position, an offset of $o = 1.2$ was used. With parameter k it is possible to influence the spread between nodes, i.e., more edges are treated of similar weight and thus the spread will become large. This is advantageous in the layout, since closely related nodes are placed next to each other but do not overlap. A value of $k = 0.2$ turned out to be the best trade-off for visualization purposes.

3.4.3 Energy based layout

A more recent and alternative approach to layout the dependencies between artifacts and problem reports is the application of an energy base clustering algorithm such as proposed by Noack [108] called *edge-repulsion LinLog energy model*. The main difference between models—such as Fruchtman and Reingold [61]—and the new LinLog model is, that edge lengths are not treated uniformly in the new model. In the LinLog model, the attraction and repulsion is exploited to build sets of related nodes.

In multidimensional scaling, as used in the previous section, distances are treated as graph theoretical distances, i.e., which is defined as the length of the shortest path between two points. Energy-based graph layout methods liken graph vertices to physical objects that exert forces on each other. Graph vertices that are connected by an edge attract, to ensure that they are placed closely. All pairs of graph vertices repulse, to ensure that non-related vertices are placed at larger distances. The resulting graph layout is an energy-minimal state of the force system. The LinLog model $U_{LinLog}(p)$ of a drawing p is defined as

Figure 3.8 Example for relationship between bugs, files, and features. The numbers indicate modification reports with associated problem reports.



follows:

$$U_{LinLog}(p) = \sum_{\{u,v\}\in E} ||p_u - p_v|| \quad - \sum_{\{u,v\}\in V^{(2)}} \deg(u)\deg(v)\ln||p_u - p_v||. \tag{3.13}$$

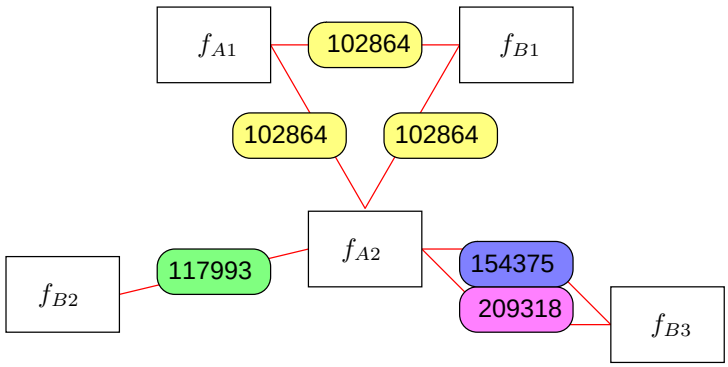
In this formula, p is a layout (i.e., a mapping of the vertices to positions in two- or three-dimensional space), $U(p)$ is the energy of p, p_u and p_v are the positions of the vertices u and v in the layout p, $||p_u - p_v||$ is the Euclidean distance of u and v in p, and $\deg(v)$ is the number of edges incident to a vertex v [27].

The first term of the sum can be interpreted as attraction between vertices that are connected by an edge, because its value decreases when the distance of such vertices decreases. The second term can be interpreted as repulsion between all pairs of (different) vertices, because its value decreases when the distance between any two vertices increases. The repulsion of each vertex v is weighted by its number of edges $\deg(v)$. Through this weighting, the second term is more naturally interpreted as repulsion between all pairs of edges than between all pairs of vertices. More precisely, the repulsion acts not between the entire edges, but only between their end vertices. So the basic ideas are that the edges in the coupling graph cause both attraction and repulsion, and that every edge causes the same amount of attraction and repulsion [27].

3.4.4 Exploiting logical couplings

EvoGraph exploits the logical couplings to identify evolutionary relevant entities and to generate visualizations as feedback for the system engineer. These visualizations are based on a graph $G(V, E)$ whereas the files represent the vertices and the logical coupling the edges. The logical couplings can be either unqualified or qualified. In the following example we use qualified logical couplings (see Figure 3.8).

Figure 3.9 Resulting graph based on the information from Figure 3.8.



From the feature extraction process we obtain the disjoint file-sets representing the functionality for a feature. Considering our example, feature F_A consists of the files $\{f_{A1}, f_{A2}\}$, feature $F_B = \{f_{B1}, f_{B2}, f_{B3}, f_{B4}\}$. They share the problem reports $P = \{p_{102864}, p_{117993}, p_{154375}, p_{209318}\}$ which constitute the logical coupling between the two features. The file f_{AB} is considered to be part of the feature fCore as outlined in the feature extraction process (see Chapter 3). The described information is transformed into the following weighted, undirected graph:

$$G = (f_{A1}, f_{A2}, 1), (f_{A1}, f_{B1}, 1), (f_{A2}, f_{B1}, 1), (f_{A2}, f_{B2}, 1), (f_{A2}, f_{B3}, 2). \tag{3.14}$$

and subsequently exploited to generate an appropriate layout. Figure 3.9 depicts the resulting graph and also a possible solution for the layout.

If a direct call relationship between two entities in the system does not exist, possible causes for logical coupling are frequently dependencies on data-types or data-structures which are inherited from other files. In our example, f_{AB} could be an include file providing some general definitions which are used by f_{A1}, f_{A2}, and f_{B1}. The challenge for software evolution analysis is to determine the impact of such dependencies onto the evolvability and to provide adequate feedback.

3.4.5 Experiment with feature fHttp

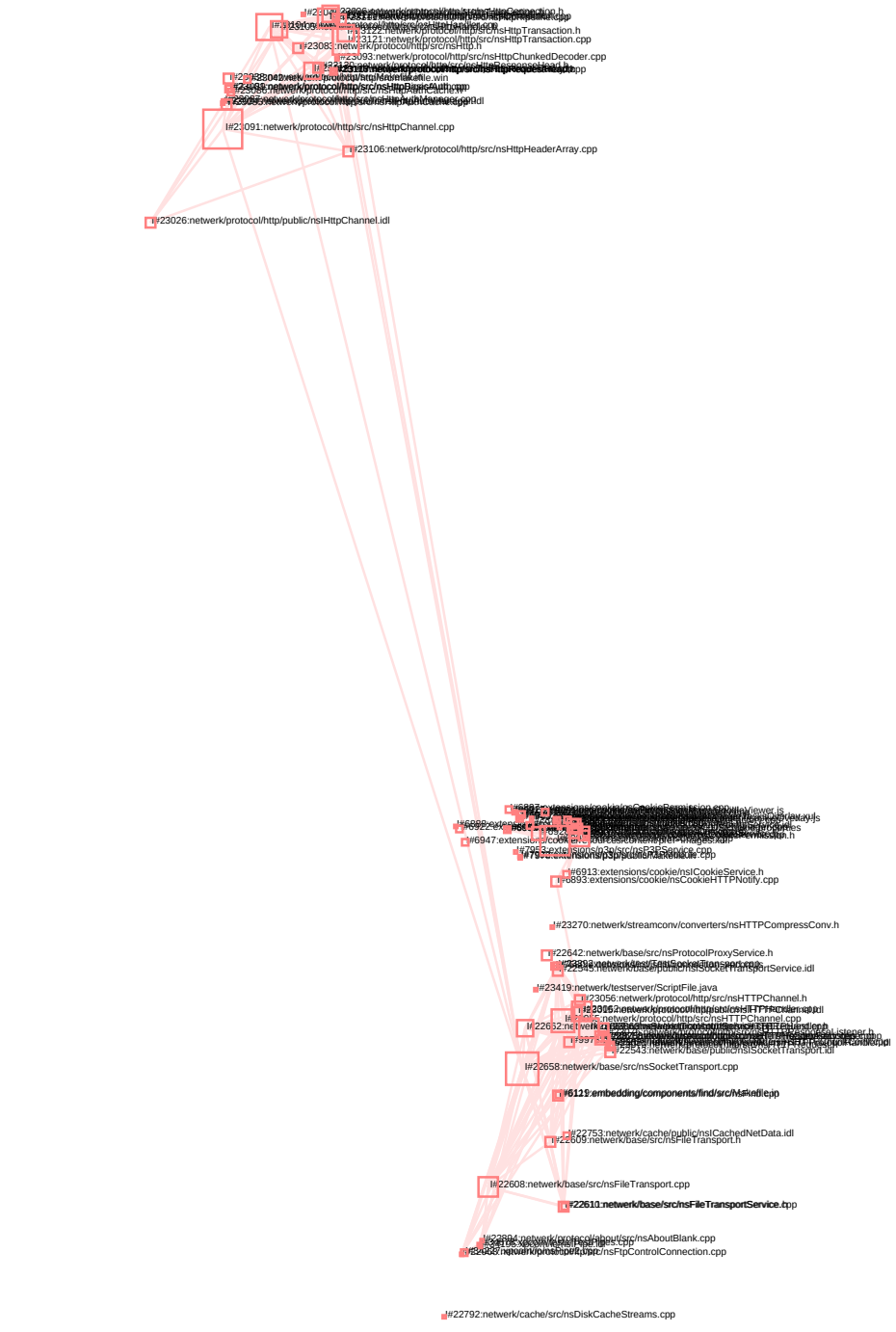
In this experiment we are interested how the *Mozilla Application Suite* will be clustered on the basis of the LinLog model. To reveal the dependencies between clusters and to observe the building of clustered, we selected all problem reports which are associated with one of the features. Thus, two files such as f_{A2} and f_{B3} are connected if they share at least one problem report. Using our example from Figure 3.9, the problem reports p_{154375} and p_{209318} cause two edges leading from f_{A2} to f_{B3}.

Table 3.3 Details for logical coupling of clusters with involvement of feature fHttp in the *Mozilla Application Suite* (excerpt). As observation period we used the years 1998–2002.

clusterid	specific reports	total reports
191	4	52
76	4	44
401	62	219
361	132	228

Since we are interested only in the results for the feature fHttp, we have identified those clusters which have problem reports associated with this feature. Table 3.3 lists the result for some clusters where we found problem reports with a relation to the selected feature. Basically two clusters are of interest. The first cluster with the $id = 401$ consists of 27 artifacts which are located below the sub-module *netwerk/protocol/http*. This reflects a good encapsulation of functionality and a clear separation of concerns. In contrast to this,

Figure 3.10 Detailed sketch of cluster 401 (upper left) and 361 (lower right).



the second cluster ($id = 361$) consists of 68 files from the modules *netwerk*, *extensions*, and some from *xpcom*, and *embedding* as well. Interesting is also the relationship between the two clusters. As depicted in Figure 3.10 the clusters are clearly separated but have a number of connecting logical couplings. For cluster $id = 361$ the file *network/base/src/nsSocketTransport.cpp* acts as "peer" for a number of files from the other cluster which are located in a different sub-modules of the *netwerk* module.

3.5 Résumé

The Extraction, preparation, import, and validation of the required information from the various sources took a considerable amount of time to obtain a *Release History Database* with sufficient data in respect to quantity and quality. Though, some of the import steps are already automated, the manual determination of information such as the mapping of software features onto source code or the inspection of problem report information would have to be redone with other systems as well. Thus, a better integration of the various information sources would ease the gain of information for evolutionary analyses.

With respect to the grouping process, the results are not surprising since we expected to find a higher logical coupling within a module (*adhesion*) rather than between modules (*stickiness*). Thus, we have to find a systematic approach to disclose the more interesting *stickiness* and to avoid the clutter caused by the logical coupling within a module aka *adhesion.*

Chapter 4

EvoZilla - Analyzing Evolving Systems

Large software system or families of related software products are complex and heterogeneous constructs. With the increased computing power, large parts of a software system are coded in higher level languages or provide extensive configuration mechanism via scripting languages. Changes affect therefore different implementation technologies used in a software systems. A small change in a configuration file may have strong impact on the overall system. Another relevant aspect in the longitudinal analysis of software systems concerns the information flow between different parts of related products. The questions which have to be answered are for instance the kind and distribution of changes from one system to another, the distribution of effects with respect to modules or components, or the existence of evolutionary hot-spots.

Thus, we developed three analysis techniques focusing on different aspects of hot-spot mining in the release history: *EvoGraph* is a systematic approach to identify relevant structural shortcomings based on information obtained from the release history. The identified entities are also evaluated for their structural stability and appearance of patterns in their longitudinal development; *EvoTrace* evaluates and compares the changes in the runtime behavior between two versions of a software system on a quantitative-level; *EvoFamily* exploits the commonalities in the release history of a family of related products and points out the strong structural relation between source code entities and documentation as well.

All techniques use the same technological foundations as described in the previous chapter such as the *Release History Database* or the grouping of related artifacts. Intermediary and final results are therefore interchangeable and allow one the evaluation of the identified entities with respect to different evolutionary properties.

4.1 EvoGraph

Structural analysis approaches frequently fall short in an adequate representation of information for longitudinal analysis. By compounding the two underlying information spaces in a single approach, the comprehension about the interaction between evolving requirements and system development can be improved significantly. We therefore use a lightweight approach based on release history data and source code changes, which first selects entities with evolutionary outstanding characteristics and then indicates their structural dependencies via commonly used source code entities. Our approach completes typical release history mining and source code analysis approaches, therefore past re-structuring events, new, shifted, and removed dependencies can be spotted easily. Furthermore, the *EvoGraph* approach combines the two information spaces evolution and structure to point out and assess structural stability, recurring modifications, or changes in the dependencies of the file-sets under inspection.

4.1.1 Approach

Our initial assumption about the characteristics of the cross-cutting change transactions in our *Mozilla* case study was, that we would find co-changes which introduce a direct or indirect structural dependency

between files having a high logical coupling. In contrast to this assumption, most of the co-changes concerned changes of indirect dependencies such as a common ancestors in an inheritance hierarchy or commonly used data types. Their creation itself within a co-change was seldom to observe. Thus, the mining approach has to take account for this observation.

EvoGraph is a systematic approach to identify structural shortcomings in large scale software primarily based on release history and small source code deltas. Our approach exploits the change history and source code information to obtain a coherent information space with historical co-change dependencies and structural information required for the assessment of a system's evolution. The underlying principles can be described as follows.

We use logical coupling to identify relevant file-pairs for further exploration on the basis of their spatial distribution. Then, in mining the longitudinal aspect of a files history, we build a structural change vector describing all changed structural dependencies. Moreover, we also record information from files appearing within the same co-change transaction. The benefit of considering also related files is the option to deduce "change semantics" related information as well. For instance, a file appearing frequently in user interface related co-change transactions could be an event handler for some GUI components. We therefore propose the following five main steps for the realization of the *EvoGraph* approach:

1. *file selection*: as starting point of the analysis process, *relevant* source files have to be selected based on data from the *Release History Database*. Here, relevant means files which exhibit an extraordinary *logical coupling* with respect to *cross-cutting change* transactions. This processing step requires the *correlation hypothesis H1a* and the *traceability hypothesis H1b*.
2. *co-change visualization*: the *stickiness*-view (depicted in Figure 5.9) generated from the selected data set provides co-change information at a glance and supports the directed search for structural shortcomings. A system engineer may select files from this view for a further detailed analysis, e.g., files which are connected with many "satellites" files. This visualization is base on the *stickiness filter hypothesis H2*.
3. *fact extraction*: for the selected files in the preceding phase, the detailed change transaction information is collected from the *Release History Database* and as result structural change vector are created for every file within a transaction. For the fact extraction we rely on the *source diff hypothesis H3*.
4. *mining*: next, the structural change vectors are the input to the *mining change transaction data phase*. Output is a description of the longitudinal development of structural dependencies of selected files.
5. *visualization*: finally, the structural dependencies with respected to their longitudinal development are visualized in an electrocardiogram (ECG) style diagram. Patterns or anti-patterns are detectable in this view as well.

In the following sections we briefly describe each of the above steps. The preparatory work and fundamentals are described in the earlier chapters: the hypotheses *H1a*, *H1b*, *H2*, and *H3* which the process is relying upon are formulated and discussed in Chapter 1, background information about the building of our *Release History Database*, and grouping of artifacts is discussed is in Chapter 3.

4.1.1.1 File selection

First, we determine a set of files which represent a defined set of the application's functionality. Chapter 3 described the details how we used dynamic analysis to identify file-sets which implement certain features. In our *Mozilla* case study, we use the basic browsing features of the software system. This approach reduces the set of 30,000 candidate files to about 1,000 which are responsible for the system's core function in terms of the selected features.

Second, we identify those file-pairs which constitute a high module *stickiness* and potentially negatively impact the system's structural stability. Currently, we use an upper limit for the size of co-change transactions to filter out "background noise", e.g., copyright modifications frequently affecting hundreds of files without having any effect on the code itself (see also Chapter 3). The limit depends on the system under inspection and has to be manually determined by reviewing the recovered transactions.

Third, from this set of candidate file-pairs, relevant files are selected via the inspection of the path which connects them. Motivation for the choice of a metric is their capability to reflect evolutionary and structural properties for the visualization: (a) file-pairs which exhibit a strong logical coupling should be

placed close together in the visualization; and (b) we are interested in coupled file-pairs which reside in different modules.

The following metric selects only those file-pairs which have a path via the *root node* of the system structure graph $G(V, E)$ (see also Figure 1.2):

$$w_{v_i,v_j} = \begin{cases} \frac{c(v_i,v_j)}{c_{max}} & \text{if } path \text{ contains the } \textit{root node} \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

File-pairs with weight $w = 0$ are not considered in the subsequent determination of co-changes. For all other file-pairs, the weight w_{v_i,v_j} is the normalized value of the number of co-changes c for the file-pair v_i and v_j. Whereas the normalization value c_{max} is the maximum number of equally weighted co-changes found which match the selection criterion. Other metrics to determine file-sets, e.g., between files within the same path of a sub-module hierarchy, can be used to assess the conditions on a local level. The metrics definition also enables the assignment of different weights to different path lengths which impacts the layout of the *stickiness*-view in the next phase. A different weighting function could be a time lense emphasizing the more recent co-changes.

4.1.1.2 Co-change visualization - stickiness-view

To find starting points for further analyses, the so called *stickiness*-view enables the directed search for hot spots in a system's module structure. Based on the graph information obtained in the previous step, we create a visualization for the *stickiness* of the resulting file-set. File-pairs with a high *stickiness*—logical coupling crossing module boundaries—are placed closely together. Unrelated files are placed further apart. As a result, the obtained information can be used by a system engineer to identify bad smells such as God-classes.

We use a visual clustering tool based on an energy model (see Section 3.4) for the layout which provides a more natural gravity layout. Edge weights are the previously determined values for w_{v_i,v_j}. Figure 5.9 shows an example for the *stickiness*-view based on data from our *Mozilla* case study. The squares in the view indicate the files and their size is an indicator for the logical coupling. Now, interesting hot-spots can be selected on the basis of this view for further investigation. If a directed search for evolutionary hot-spots is not required, the logical couplings are processed in order from the highest to the lowest number in the next processing step.

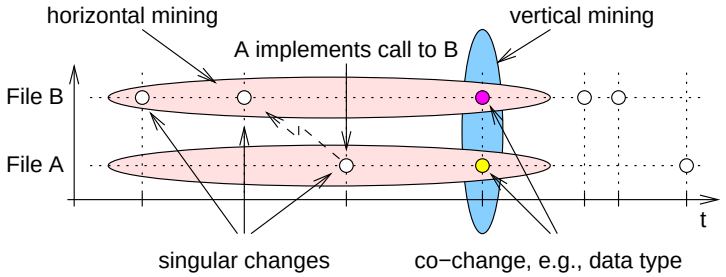
4.1.1.3 Fact extraction from code changes

The objective of this step is the extraction of facts about possible structural dependencies. To retrieve the required source code deltas we determine the respective revision number and its predecessor version by examining the *Release History Database*. Next, with the revision number information the respective source code deltas can be retrieved from the repository. They consist of different fragments and represent the parts of the source file which has been modified. Since a full-fledged parser can not be used to parse the fragments, we implemented island-parsers [106] for the different flavors of programming languages. They recognize important constructs such as class declarations. Furthermore, they are able to handle incomplete statements such as partial changes of control constructs or open comments.

Results of the parsing process are strings representing facts about the inserted or deleted code. Since only the changed portions of the source code are considered for fact extraction, it is not always detectable if a dependency has been removed completely. Unchanged portions of the source code may still maintain a dependency. Such examples are "dangling" include files where the include directive has not been removed. As a consequence, *added* or *removed* dependencies may not be accurately detectable. The following facts are extracted:

- *added provides*: a new code sequence which provides some functionality to others, e.g., class, interface or method declaration, has been detected;
- *removed provides*: opposite of the above, i.e., some declaration has been removed from the source code;
- *added depends-on*: a new structural dependency, e.g., inheritance, method call etc., has been detected;

Figure 4.1 Vertical and horizontal mining.



- *removed depends-on*: opposite of the above, i.e., a structural dependency has been removed from the source code;
- *added uses*: a string which does not represent a provides or depends-on fact has been detected;
- *removed uses*: a string has been removed; and
- *updated*: a string appears in the added and removed section of the same source code transaction.

To distinguish frequently used strings such as function or variable names from rarely used strings, we use a weighting scheme based on their overall appearance in all source code deltas. The assumption is, that unique or specific strings of a class appear more rarely than common names such as the well-known C function *printf*. From the list of extracted strings and their number of appearances we first compute the arithmetic mean of all strings detected:

$$\overline{s} = \frac{1}{n} \sum_{i=0}^{n} s_i. \tag{4.2}$$

Then for each string in the list we can compute its relevance

$$r_i = \frac{s_i - \overline{s}}{s_i + \overline{s}} \tag{4.3}$$

yielding results in the range $[+1.. -1]$ whereas $+1$ is interpreted as irrelevant, i.e., string appears very frequent, and -1 as relevant. The variable s_i in the above formula denotes the number of appearances of the respective string.

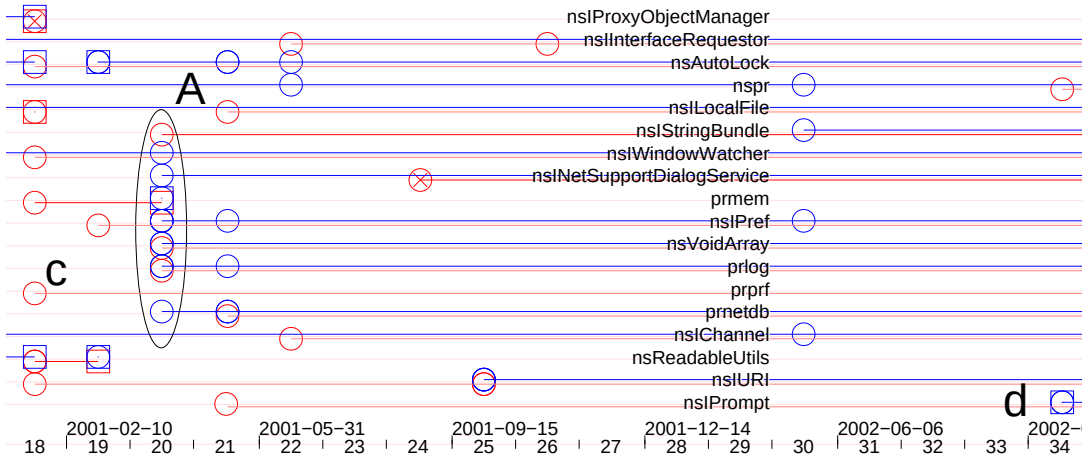
For every change transaction of a file, the extracted strings from the source code fragments are aggregated into a structural change vector. These vectors are then used to identify structural dependencies within a transaction. Another interpretation is the effect on the longitudinal development with changing dependencies. The changes may lead to patterns which are an evidence for limited agility of the system with respect to evolutionary changes. In this fact extraction step, changes which are applied to branches of the source, are not considered. If these changes were relevant for the reference architecture, they would have been merged into the main trunk of the revision tree anyway. Change transactions which affect branches, can be identified easily via their revision numbers.

4.1.1.4 Mining of co-change transaction data

Many of the analysis approaches proposed so far observe the evolution of files with respect to co-changes *or* observe the evolution of a files with respect to structural changes. However, we combine both approaches to provide a holistic view on a system's evolution (see Figure 4.1). This facilitates the integration of evolutionary and structural analysis into a single approach. The existing separate approaches can be classified as follows:

- to identify logically-coupled files and filter interesting file-pairs (*vertical mining*); and
- to find information which links timely offset changes of the respective files (*horizontal mining*).

Figure 4.2 Example for the label-view in EvoGraph.



We refer to *vertical mining* as the exploitation of co-changes when the files of interest have been part of the *same* change transaction. We refer to *vertical mining* as *the exploitation of co-changes when the files of interest have been part of the* same *change transaction*. Other change transactions such as singular changes are usually not considered. The objectives are manifold and comprise the clustering of files, pointing out structural shortcomings, or predicting source code changes. Approaches which fall into this category are described in [27, 146, 149].

The focus of *horizontal mining* approach is *the reconstruction of the longitudinal development of a single file with respect to structural dependencies*. Here, the exploitation of co-change properties is of minor interest. Proposals for such approaches are in [94, 142, 143]. This mining approach can be best characterized by the extraction of metrics and structural information for every revision of a file and reasoning about it on a per file basis.

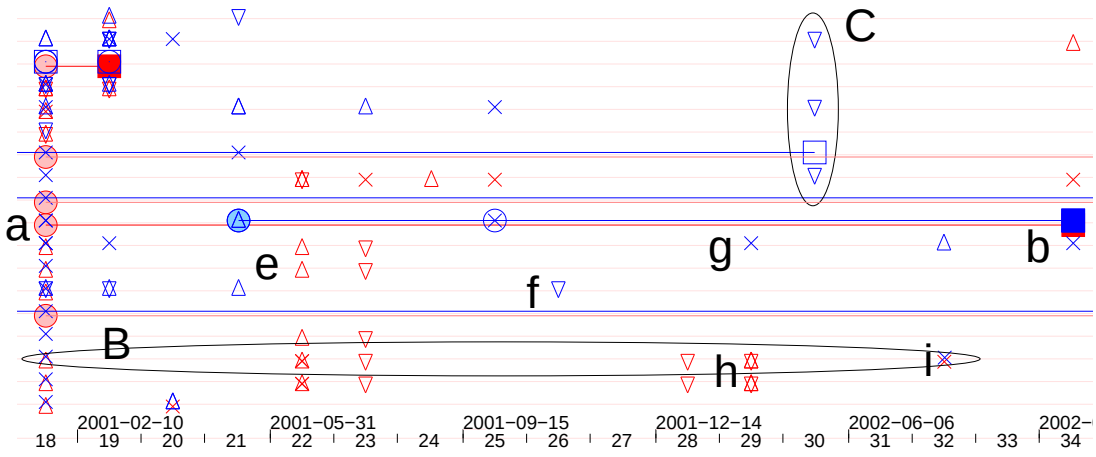
A motivating example to combine both approaches is depicted in Figure 4.1. File B for instance implements a function β which will be used later on from a function α in file A. The required change transactions may happen independently of each other but in the correct order to maintain the integrity of the source code. At some later point in time, a data type change requires the modification of the method signature. Thus *callee* and *caller* have to be modified at the same time. This modification emerges then as co-change transaction. In our approach, we use vertical mining to determine the logical coupling between file-pairs. Based on facts obtained via horizontal mining, we search for evidence in the change history of the selected file-pair to find the type of structural dependency and to model their longitudinal evolution. A benefit of our approach is, that we use source code deltas for fact extraction. Thus, the minimum effort to parse the code of all source code revisions is $O(n)$. In the worst case, when the file is completely modified in every revision, the effort can be approximated as $O(nm)$, whereas n is the total length of the file and m is the number of revisions.

4.1.1.5 Visualizing evolving dependencies

The purpose of the *EvoGraph* diagrams is the provision of visual information for uncovering change patterns, spotting commonalities, monitoring of ongoing activities, and validating past re-engineering events. Figure 4.2 and Figure 4.3 depicts the common symbols (one per line) and their respective changes (glyphs) of the features fHttp and fHttps. For every source code change we build the tuple of structural change vector and time which represents a *change event*. In *EvoGraph* the detected change events are classified with respect to their information extraction method and display in two different (sub-)views:

- *label-view*: provides the more reliable information about structural dependencies and is based on the *depends-on* and *provides* facts. An example for this view is depicted in Figure 4.2; and

Figure 4.3 Example for the word-view in EvoGraph (symbol names not shown).



- *word-view*: the *uses* and *updated* facts are the information basis for this diagram. Here, the strings are only statistically weighted and they do not necessarily imply a real structural dependency, i.e., the same *word* can be used in different files for similar purposes (see Figure 4.3).

To depict the modification events with respect to the two categories we use colors and glyphs (see also Figure 5.10 in Chapter 5). The color red (▬) indicates change events of one file-set and the color blue (▬) indicates the events of the other set. The following glyphs indicate the different events recorded for the common structural dependencies (one per line):

- in a source-code modification of a file a *new provides* has been detected, e.g., a class declaration or interface definition, which is depicted via the glyph ● (a);
- a *removed provides* has been detected ■. As a consequence, the interface, method or variable is no longer usable by other entities of the software system (b);
- a *new depends-on* represents a reference onto something new, e.g., usage of class within another declaration, has been added to the source code ◯ (c);
- a *removed depends-on* reference has been detected □ (d);
- in a source-code modification a relevant *new uses* has been detect △ (e);
- with the *removed uses* a word has been removed from source code ▽ (f); and
- when a word has been *updated* the event is depicted as × (g).

Due to the aggregation of several modifications within one time-slot, some of the glyphs overlay and build "new" glyph types, e.g., a *word* has been added and removed again within one time-slot ✡ (h) or modified in two files within the same time-slot ✳ (i). Other change events may overlap as well and form different glyphs. When it is possible to track a logical and timely connection between *labels* or *words* the respective events are connected via a solid line, e.g., added and removed provides ●—■ (a - b), to indicate the continuous dependency.

The visualizations inform about major additions of dependencies (A), frequent changes or updates (B), and also about removal of symbols (C). The following described analysis uses the visualized events to identify patterns—and more important—anti-patterns in a system's longitudinal development.

4.1.2 Analysis types

With the visualization of the selected files, the tracking of their evolution is basically completed. Next follows a description of different analysis approaches which provide deeper insight into the evolutionary processes of a software system. The analysis is part of the mining activities discussed earlier.

4.1.2.1 Patterns and anti-patterns

With respect to the visualizations we define patterns and especially anti-patterns which point out structural shortcomings of entities with curious longitudinal behavior such as growing or change rate. In [92] Lanza defined a number of patterns which we will use subsequently. For some of the defined patterns such as *supernova*, *white dwarf*, *stagnant*, or *persistent* we did not find an applicable scenario in our longitudinal analysis of structural dependencies. Instead we introduce three new patterns: *black-hole*, *skip-jack*, and *daisy-chain*. Especially the occurrence of anti-patterns may indicate bad design, lack of knowledge about the actual architecture or other shortcomings such as inadequate consideration of system evolution. In all cases such anti-patterns are costly and should not appear in well-design systems. While the first of the following patterns refers to the *stickiness*-view, the other patterns refer the to the longitudinal view:

- the *read giant* or *God-class* binds a number of other files to its surroundings which build a group with respect to their logical couplings. God-classes are detectable in the *stickiness*-view. The elements of the cluster can be determined via their distance from the God-class which is in the center;
- in the *black-hole* pattern several structural dependencies are removed simultaneously from the source code. After this successful re-structuring or code *clean-up* event new structural dependencies concerning the removed dependencies should not reappear;
- the *skip-jack* anti-pattern appears after a *re-structuring* event were a structural dependency has been removed. Different reasons may require that the same structural dependency has to be included again;
- the *day-fly* anti-pattern indicates a structural dependency which is introduced and removed again in the same time-slot or after a small number of revisions. It is basically the inverse of the skip-jack anti-pattern. A day-fly can be detected via its short-lived existence of a few revisions. There are no other usages before or after the day-fly associated with this dependency. Day-flies indicate sub-optimal structural changes which had to be modified again at a later point in time causing extra effort for modifications and testing;
- a *pulsar* anti-pattern characterizes the appearing and disappearing behavior of a dependency. This pattern may indicate debugging activities, requirements and design problems, or unclarity about the implementation of some functionality; and
- in the *daisy-chain* anti-pattern a structural dependency is first used by one file and later exclusively used by a second file. This pattern indicates that the first implementation of a solution had some drawbacks which had to be superseded by a more carefully designed one.

Examples of the *EvoGraph* visualizations and the detected patterns with respect to the *Mozilla Application Suite* are provided in our case study (see Chapter 5).

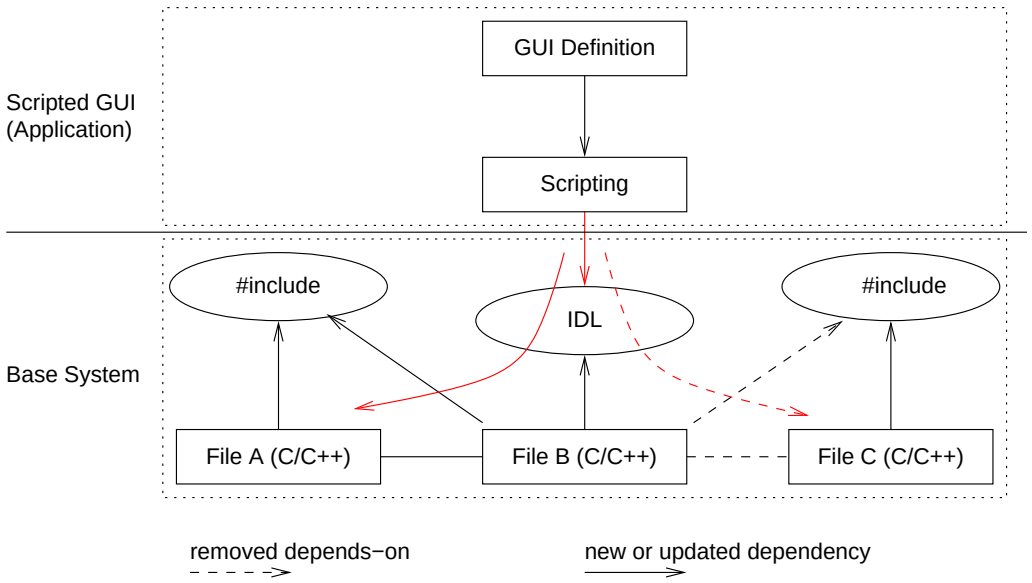
4.1.2.2 Structure evaluation

Objective of the structural evaluation are the identification of the direct and indirect dependencies which hamper a systems evolution. With the lightweight fact extraction process of *EvoGraph* based on file-level, two basic analysis types can be implemented:

- *commonality analysis*: two files or two file-sets are analyzed for common strings in their longitudinal development. Only symbols common to both sets are selected. This makes views of this type interesting for dependencies analysis. All subsequent *EvoGraph* views and concrete results about structural stability and evolutionary patterns are based on this analysis type.
- *aggregation analysis*: for a single file or a file-set all changes with respect to their longitudinal development are depicted. Assuming that source code is basically added to file and seldom deleted, the result is an aggregation of all the symbols used. A distinction between local and shared symbols is not made, but the assessment of structural stability and the detection of evolutionary patterns is also possible. This analysis type is not exploited in *EvoGraph*.

Besides the two main types described above, the analysis can be further divided with respect to the analyzed artifacts. The analysis granularity relies basically on the natural unit of files, though other unit types would be advantages especially for feature analysis.

Figure 4.4 Schematic representation for extracted dependencies of files in a co-change transaction.



- *file*: two arbitrary files or file-sets are analyzed for their structural dependencies and evolutionary patterns. In our approach we are interested in comparing those file-sets which were identified to have a high logical couplings across module boundaries;
- *module*: as modules represent different file-sets, they can be analyzed easily. Since the number of files can be high, a preselection of the more relevant candidates will be required to avoid clutter of the resulting views; and
- *feature*: two sets of possible related features are analyzed for their *interwovenness* with respect to their direct or shared structural dependencies. As result the identified structural entities such as interfaces, methods, or variables provide clues about possible feature interactions.

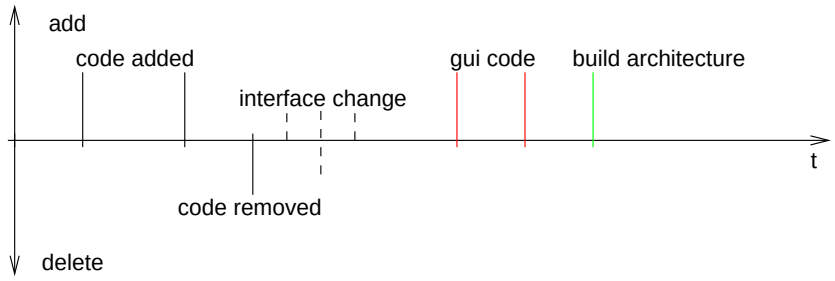
On top of the proposed analysis types, further analyses can be implemented. Examples are the assessment of structural stability, analyses with respect to the improvement of adhesion and reduction of *stickiness.* Also prediction models can benefit. Providing an advice about resolving structural shortcomings is an asset in planning future re-engineering activities. After success re-engineering, the appearance of evolutionary patterns are indicators for possible unsuccessful attempts to resolve these conflicts.

4.1.2.3 Change type

Fine-grained change analysis offers another interesting property with respect to reasoning about longitudinal changes and their causes. Within a limited range it is possible to derive *change-type* information—similar to a change semantics—about the co-change transactions. We consider a *change-type* as *the "semantic" information obtained from file type analysis.* For our purposes it is sufficient to analyze the suffix of a file-name such as *.idl* (for component interface) or *.xul* (for user interface) to deduce the change-type. As we analyze all files within co-change transactions we are able to characterize the "semantics" of the transactions. A file frequently appearing in transactions with files related to user interface changes certainly plays a different role than a file appearing in transactions with files related to processor specific assembly code.

Figure 4.4 depicts an example with a dispatcher file (File B) responsible for handling application events (such as *nsCSSFrameConstructor.cpp* in our case study). Based on the co-change transactions it is possible, for example, to deduce its involvement with GUI related changes. Though, File A is not directly invoked from the scripted GUI, we possible could find co-changes with File B (vertical mining) in the release history. For files appearing in such co-change transactions we may conclude that they change concerned

Figure 4.5 Example for schematic representation of a file's evolution with respect to change type events.



functionality related to some visual representation. Via horizontal mining we are then able to identify the actually affected structural dependencies.

Thus, we are aiming at combining the information obtained from the structural change vectors of a file with the change-type information into a single evolution vector. The vector represents the information substrate gathered from all change transactions and describes a file's evolution also with respect to "semantic" changes. For instance, the evolution vector could comprise the following information derived from the file-name extensions mentioned previously:

- *cross-platform change*: the change transaction affected files which are specific to the targe hardware- or software platforms. This may break the platform independence of the application and requires extra testing effort;
- *user interface*: code for a feature or function has been added, modified or removed which implies that code on different abstraction layers has been modified; Consequences are high testing effort, inherent problems with feature interaction, or new bugs due the large code changes. The extra effort for implementing a feature in a late project phase my increase by an order of magnitude [40];
- *component interface*: a component interface has been added, changed, or removed. Consequences are changes of the component model of the application;
- *build architecture*: files were added or removed from the corresponding application build files also known as *makefiles*. Consequences are new architectural dependencies (e.g., call, access, inheritance, etc.) which should be reflected in source code changes as well;
- *signature*: replacement of a function or variable type or interface with a different one. Consequences are large change transactions with many modifications scattered in the respective source files; and
- *code* added, changed, or removed: the change transaction introduced new source code or parts of the source has been modified. Changes to public interfaces or the build architecture were not required.

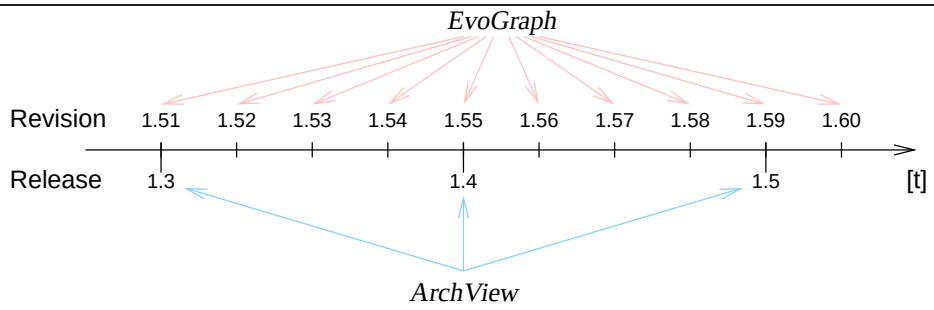
These classification types add the change-type information together with the added or removed dependencies to the structural change vector obtained from the *fact extraction* phase. The classification supports the system engineer in reasoning about the impact "weight" of a co-change coupling on the structural dependencies within in the system.

A file having many different change-types over a significant period of time may indicate a violation of the separation of concerns. A certain percentage of user interface related changes, for instance, might suggest to review the implementation of the Model-View-Controller (MVC) pattern [65]. Figure 4.5 depicts a sketch of such an evolution vector.

4.1.3 Data connectivity

Important for the usability of an analysis approach is its integration with other analysis environments. *EvoGraph* and its visualizations is designed to run in standalone mode but results can be integrated into other structural analysis approaches such as *ArchView* [111]. While *EvoGraph* analyzes the source code changes of every revision and therefore *all* structural changes are tracked, in more structure related approaches usually only selected releases are analyzed. Considering frequently modified files which are modified ten times

Figure 4.6 Integration of evolutionary results of *EvoGraph* with structural results from *ArchView*.



per month on average or even more, then the arbitrary selected source code releases would make it difficult to identify patterns in a file's longitudinal development.

The integration of both approaches has to happen per file on the revision with the largest revision number having a timestamp lesser than the given release date. In properly maintained repositories, this information is directly available via the tagged source code revisions. Figure 4.6 depicts how the fine-grained analysis process of *EvoGraph* can be synchronized with the *ArchView* process. The synchronization is required since *EvoGraph* uses revision-based information and *ArchView* for instance uses release-based information derived from the complete source code packages.

Prerequisite for the synchronization is either that the revision numbers are known also for the *ArchView* dataset or the revision numbers can be deduced from the recovered time-scale. Then, with the identified synchronization points, a symbol on can be unambiguously located in the source code via the three-tuple:

$$SI = (\text{file-name}, \text{revision-number}, \text{line-number}). \tag{4.4}$$

Our experience is, that especially with C/C++ code a number of differences in the notation of functions and methods between different tools exist. The usage of the three-tuple is therefore simpler than the comparison of method signatures as their representation may vary.

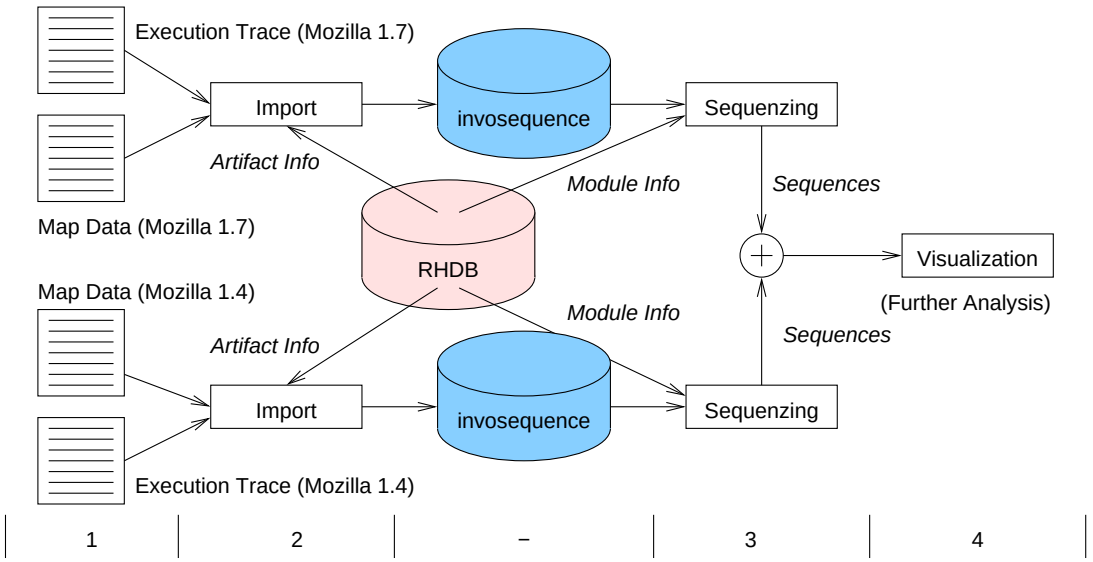
The benefit of intertwining both approaches is that complete structural analyses are computationally expensive and therefore they can not be run on every single update. Such complete analyses are for instance useful with the major releases of the software system. Once synchronized, our lightweight analysis approach offers the opportunity to track all intermediary change events. Figure 4.6 depicts an example with releases `1.4` and `1.5` as the basis for *ArchView*. The timely distance between releases to inspect depends on a number of different factors such as development activities and should be chosen as required. A half year distance is sufficient in most cases. Another scenario is the observation of all events which occur after a detailed analysis of the last release. As depicted in Figure 4.6 with release `1.5` in our example as the basis. This enables the extrapolation of the obtained *structural fact base* via *ArchView* and enriches it with longitudinal information via ongoing monitoring. Via tracking of insertions and deletions of source code lines the positions of the symbols can be tracked easily, even if no structural data about new symbols are available. These new symbols with no structural counterpart have be integrated later into *ArchView*.

4.2 EvoTrace

In this section we describe the methodology used in our approach called *EvoTrace* to exploit program execution traces for retrospective software evolution analysis.

Dynamic analyses based on execution traces are used in software testing, software performance analysis, distributed and parallel systems evaluation, and to some extent also in program comprehension and re-engineering to facilitate the understanding about interactions between building blocks of a software system such as modules. While in structural analysis dynamic information such as call graph information is used to get a complete *static* picture of the actual implementation of a software system, execution traces have not been exploited for detailed retrospective software evolution analysis.

Figure 4.7 Import and analysis process of *EvoTrace*.



Execution traces produced from instrumented code reflect a system's actual implementation. This information can be used to recover interaction patterns between different entities such as methods, files, or modules. Some solutions for the detection of patterns and their visualization exist, but are limited to small amounts of data and are incapable of comparing data from different versions of a large software system.

Our *EvoTrace* approach recovers module views which facilitate the comprehension of each module's evolution with respect to their effect on run-time data. The methodology allows us to track the longitudinal changes of particular modules and present the findings in three different kinds of visualizations. For the implementation of the *EvoTrace* approach within the *EvoZilla* framework we extended our *Release History Database* with the *invosequence* table to accommodate also the execution traces. Linkage with the existing entities in *cvsitem* enables the access of dynamic information from a structural analysis approach.

4.2.1 Approach

Most of the information recorded in execution traces is captured also by profiling information. Thus, profiling information can be used to generate a call graph or to gain information about the invocation frequency of each method. However, patterns of invocation are not recorded, i.e., it is not possible to deduce how frequently a method C was invoked as $A \rightarrow C$ or $B \rightarrow C$.

As a further shortcoming, we identified the impossibility to determine how these invocations are distributed over the execution time, i.e., during which program execution phase the invocation patterns emerge. Reason for the limited data recording capabilities of "traditional" profiling is the information explosion during program execution and the impact on execution time if detailed data are gathered. Nevertheless, for a significant number of software systems and their use cases, these limitations can be neglected if data can be collected using specific test environments.

In contrast to a call graph analysis of a single release of a software system, in retrospective software evolution analysis we are interested in the modifications applied to the software system which describe the changes from one release to another. More specifically, we are interested in the occurrence of specific invocation patterns between modules or files and their change when different releases of a software system are compared. One challenge with dynamic data is its size: simple scenarios can result in very large execution traces. Because of that, researchers have investigated compression techniques to cope with the size challenge.

EvoTrace comprises four steps as depicted in Figure 4.7. The first two steps are already used in the feature analysis, whereas the last two steps are specific to *EvoTrace*:

Table 4.1 Representation of invocations in the *Release History Database.*

Name	Size	Description
id	4	Unique ID for this event
callee	4	Text segment address of called method
caller	4	Call issuing address
type	1	Method enter e (call) or exit x (return)
threadid	1	ID of thread context
level	1	Invocation or recursion level
involibid	1	ID of software library
cvsitemid	3	ID of artifact in *Release History Database*
invofuncid	3	ID of invoked function
sourceline	3	Source code line number

- instrumentation, trace- and map-data generation;
- import from execution traces;
- sequencing invocations between modules; and
- visualization.

While the first step is development platform dependent, the subsequent steps of *EvoTrace* use Perl, Java and MySQL which are available for a number of OS platforms. Next, we describe the data representation and import process based on our implementation of *EvoTrace.* Though, the process is tailored for our Linux development environment, other operating systems can be used as well, provided that the required trace information is available. Again, central element is the *Release History Database.*

4.2.1.1 Instrumentation, trace- and map-data generation

As noted by Hamou-Lhadj in [77] three methods exist to generate traces of method calls: insertion of probes such as prints; modification of the runtime environment such as Java; and debugging to monitor program execution. The first method is supported by the GNU compiler collection (GCC). Thus only two functions—one for entering and one for exiting—must be implemented. Appropriate calls to these functions are then generated by the compiler. After compiling and linking, the application can be tested. For *Mozilla* we used a typical scenario in which a web page from our web server was loaded. To avoid user interaction with the application, the running program is terminated via an external `QUIT` signal when no more additional events are recorded.

Before the trace information can be used in the further analysis process, the recorded memory addresses must be mapped onto method- and file-names. This is done with map data generated from the two GNU tools `ldd` and `nm`. The first tool, `ldd`, generates a mapping of base addresses for the dynamic linked libraries. These base addresses are required to determine the library for which a call was recorded. The second tool, `nm`, lists symbols from object files with source file name and line number information. Both outputs are written to a map file so the mapping information together with the trace data can be used in the following import process.

4.2.1.2 Importing execution traces

Result of the import from the execution traces and map file data are two separate database tables containing the respective traces of each *Mozilla* release with linkage to existing artifacts in the *Release History Database.* The import is divided into two phases and accomplished via a Perl script:

- read map file information and try to find corresponding artifacts in the *Release History Database*; and
- read the execution trace information and add one record in the database for each event in the trace file.

After some experiments with the trace data we decided to use the format depicted in Table 4.1. The field sizes are specified in bytes. The trace data, generated during execution of the testee, are stored in the four fields: *callee*, *caller*, *type*, and *threadid*. The remaining fields are evaluated during the import by the Perl script:

- *id*: a unique identifier for each event assigned during data import;
- *callee*: the code address of the method invoked during program execution from *caller*;
- *caller*: this address determines the originating point of an invocation in the execution trace; While the callee address has a direct mapping to linker addresses, the caller address maps to the code segment between methods and thus is not directly usable. Instead, an application of the caller address lies in the search of corresponding *enter-exit* pairs. These pairs can be identified unambiguously within a thread context via the 3-tuple *callee-*, *caller*-address and invocation-*level*;
- *type*: each event in the database is marked either with 'e' for enter or 'x' for exit of a method;
- *threadid*: every event requires information about the corresponding thread context (otherwise traces are intermixed);
- *level*: the invocation-level information is simply derived from the *type*-field by counting enters and exits on a per-thread basis. This information is added to simplify database queries;
- *involibid*: the instrumented application may consist of several third-party libraries. The IDs have to be considered insofar that it is required to distinguish application local calls from calls to the third-party libraries;
- *cvsitemid*: from the import of release history data into our *Release History Database*, a mapping from source files to unique IDs already exists. With the symbol information from object files we are able to map the *callee*-address to the corresponding entry in the *Release History Database*. This information is required to assign the file information to modules;
- *invofuncid*: during the import the *callee*-addresses are mapped onto unique IDs which corresponds with method signatures. These signatures are derived from the object list data provided by the fact extraction tool; and
- *sourceline*: the source code line number of the invoked function denoted by *callee*-addresses.

After importing and linking relevant information, the invocation database is ready to serve queries. In Section 5.6 we provide some examples for a quantitative evaluation. Next, we describe an analysis algorithm for the detection of interactions between modules based on the invocation sequence data.

4.2.1.3 Sequencing

We focus on invocations between different modules. This reduces the amount of information to be displayed and characterizes the communication between modules. Consequently, we are interested in invocation sequences S_1, S_2 between modules M_a, M_b and their methods or functions $f_a(), f_b(), f_c()$ such as $S_1 = M_a.f_a(M_b.f_a(); M_b.f_b())$ or $S_2 = M_a.f_a(M_b.f_a(M_b.f_b()))$. S_1 exhibits two module switches since the control flow always returns to M_a. S_2 exhibits only one module switch. These sequences are derived from the recorded data using the fields `type`, `cvsitemid` and `level`. Since data are not represented as graph in the database, we need to traverse all invocations which is performed by a small Java program. Figure 4.1 shows the (simplified) Java code which is used to detect the invocations between modules. To reveal the transitions between the different modules a data structure holds information about the invoked modules. For each change of invocation-level, the event pairs *(new event in trace* and *old event on stack)* are compared. A change is checked and recorded via the function `save_diff_module(o,n)`. This function compares the module IDs and counts the transitions within the program flow. Transitions within a method, i.e., on the same invocation-level, are recorded with the code in the else-branch. Here, the topmost element of the stack is replaced with the new event. Then the two elements on the stack are checked for different module IDs. While the if-statements check for invokes and returns such as $M_a.f_a(...M_b.f_a(); ...)$ the last else branch detects a series of invocations such as $M_b.f_b()$ and $M_b.f_c()$ in $M_a.f_a(...M_b.f_a(); ...M_b.f_b(); ...M_b.f_c(); ...)$. Every detected transition (i.e., their respective module ID) is written to a separate database table. Next, input data for visualizations are generated from this information.

Listing 4.1 Java code for transition detection.

```
Event[] events = new Event[MAX_STACK];
events[0] = trace_data ();
int cntevent = 1;
while ( more_trace_data ()) {
  Event n = new Event( trace_data ());
  Event o = events[ cntevent-1]; // get old event
  if (n.level > o.level ) {
    save_diff_module (o,n);
    events[ cntevent++] = n; // save new event
  }
  else if (n.level < o.level ) {
    save_diff_module (o,n);
    events[--cntevent-1] = n; // replace old event
  }
  else { // same invocation level
    events[ cntevent-1] = n; // replace old event
    o = events[ cntevent-2]; // get ''new'' old event
    save_diff_modules (o,n);
  }
}
```

4.2.1.4 Visualization

Since visualizations are appropriate means to recognize trends and to compare the results, we combine the obtained information from the two versions of execution traces into different diagram types. One major problem for visualization are the deficiencies of the often used Gantt charts for the presentation of $2 \cdot 10^6$ transitions between modules within the usual viewing range. Consequently, we need to reduce the amount of information. A frequently used solution is the application of sub-sampling. Since no constraints on the time-slots are given, we use twenty time-slots since it is most appropriate for use in the following three diagrams.

- a Gantt chart provides a good view on different phases of the program execution. Different phases such as system initialization or user interface related activities can be distinguished;
- the "matrix" view emphasizes the quantitative aspect of changes in invocations between modules. The two communication directions between entities are depicted separately; and
- for a more detailed view on the interaction between modules we use Kiviat diagrams. In this view, the communication between one module with respect to other modules is shown. A separate axis in the diagram is used for each of the selected modules.

Based on this sub-sampling interval, we count the module transitions detected in the previous step and generate the data sets automatically via a Perl script. The results are depicted in Figures 5.17, 5.18, and 5.19, respectively.

4.2.1.5 Optimizations

Next, we discuss some optimizations which we identified during the development of this approach and in relevant literature.

- trace data compression: as pointed out by Hamou-Lhadj and Lethbridge [77], a limiting factor is the problem of size explosion. Size reduction through pattern matching seems to be the most appropriate solution to this problem. Deactivation of instrumentation—as sometimes proposed—for certain files to reduce the amount of generated traces would require detailed knowledge about the software system to inspect because otherwise important invocation transitions could be lost. Another drawback of the

deactivation solution is the required effort to manually enable or disable instrumentation on a per method basis. To reduce the amount of records the currently separate *enter* and *exit* records can be merged since most of the information is redundant. Aside from the size reduction extra lookups to find a corresponding invocation pair are avoided.

- standard database technologies support fast access to events of selected modules: if the analysis environment provides sufficient computing power and memory ($\geqq$ 3GHz Pentium 4, $\geqq$ 1GB), database tables can be kept in memory. Thus the execution time for entries in ad hoc queries ranges from fractions of a second to less than one minute depending on whether an index can be used to resolve the query.
- support for detection of sequences and patterns: one field of future work is the detection of invocation patterns. Detected patterns are a prerequisite for the implementation of a fast comparator function of the version related trace data.
- handling multiple versions of execution trace data: in *EvoTrace* we use different database tables to handle the execution traces originating from different version of the test program. Inclusion of the version information into the tables would create a large amount of redundant information.
- linkage with existing release history information: this linkage is required to facilitate the evolutionary and structural analysis process. The benefits of combining these processes have been addressed by the *EvoGraph* approach which we discussed in the in the previous section.

4.3 EvoFamily

Unanticipated evolution of a single software system enforced through changing requirements can lead to diversification and will result in different closely related products. These related products require a high maintenance effort which could be avoided by building a platform for a product family from existing software assets. To identify assets from related products which can be used as basis for a product family, retrospective software evolution analysis can help to point out artifacts which exhibit a strong change dependency. We therefore propose an approach called *EvoTrace* to identify and out-line possible information flows between related systems. With respect to the existing infrastructure, the following information and analysis steps have to be performed:

- extend our *Release History Database* for extracting change history information from a family of related products and import relevant information into a set of linked *Release History Database*s;
- compare product variants on quantitative-level for a coarse assessment of the historical development and assessment of the repository information for further research; and
- apply our approach for the visualization of logical coupling between features onto change dependencies within a product.

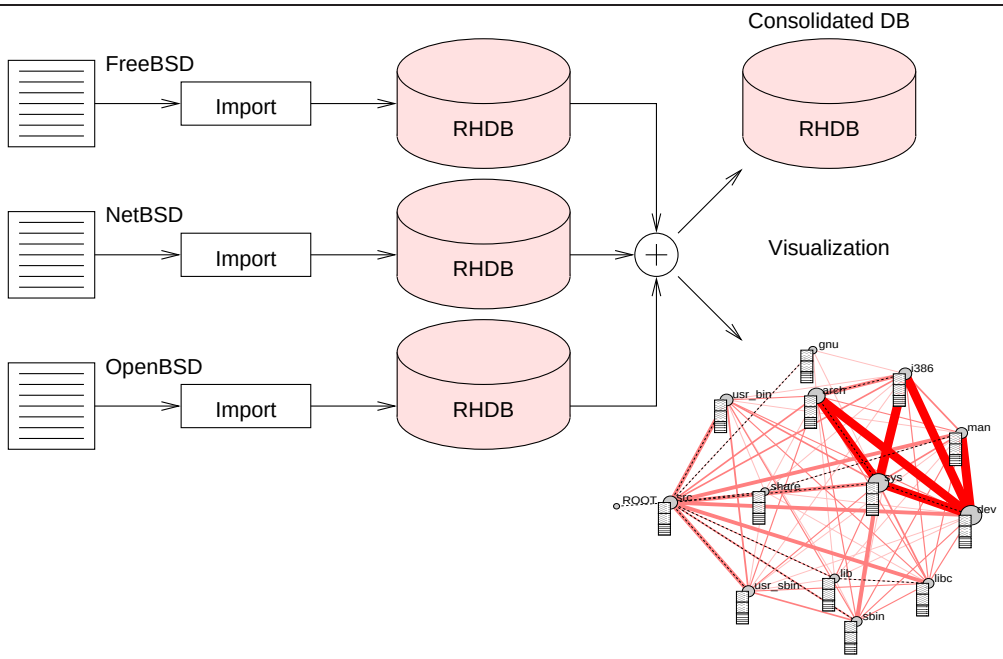
Objective is to identification of commonalities between different variants of a product line. An exploitation of these commonalities with respect to their temporal and spatial distribution provides an interesting view on the past and possible future development of the product variants.

4.3.1 Approach

Most of the proposed mining approaches such as Zimmermann et al. [149] for mining the change history or Collberg et al. [38] for visualizing a systems evolution are justified to analyze data from a single source and would therefore require adaption to support data from multiple product variants. Analyzing a single product variant implies a strict order on historical information such as check-ins into the source code repositories. In contrast to this, multiple product variants can be roughly characterized through arbitrary and asynchronous release dates, unanticipated information flow between variants, different development goals and requirements. Given these constraints, with our *EvoFamily* approach we address the problem of handling multiple, *asynchronously* maintained version control systems to identify change dependencies through “alien” source code.

Artifacts with a strong change dependency often have architectural dependencies as research by Briand et al. has shown [31, 32]. Another prevalent reason is duplicated code through *copy’n paste*. For the

Figure 4.8 Process outline of *EvoFamily*: results are a the consolidated *Release History Database* and visualizations depicting structural dependencies.



analysis of such change dependencies it would be beneficial if existing approaches and techniques can be adapted and reused to study their impact onto the module structure.

To answer the research question of source code propagation within a product family we have adopted our earlier approach for building the *Release History Database* and visualization of evolutionary information of large-scale software. We therefore propose the process depicted in Figure 4.8. Since all data sources must undergo the same preprocessing steps—log file extraction, import into the *Release History Database*, detection of change couplings—we use separate databases to store the results. For subsequent analysis transactional data from the separate databases are filtered and merged into a new *consolidated* database which is better suited for queries spanning multiple product variants. Currently we use modified variants of existing queries to gather data from the three product databases to compare them on a quantitative level. Another approach to compare system characteristics is by visually comparing graphs describing a systems history. We use a module graph indicating the impact of change dependency and their distribution with respect to different product variants onto the module structure of a single system.

In previous studies it was possible to use the release dates of the system under study as input for time-scale information. Since the BSD variants are developed independently, an artificial, common time-scale has to be created. This ensures comparability of the different system histories. Disadvantageous is that is not possible to examine and compare the processes between the release dates, since the release intervals of the different product variants are crosscut at arbitrary points. Therefore, we will use a fixed time-scale with an interval of one month regardless of the actual release dates.

To detect and relate information flow between BSD variants we decided to use lexical search in the change logs to find hints for information flow from other systems into the system under inspection. Alternatives to a pure lexical search are clone detection in source code, comparison of the structure of changes, or advanced indexing and text analysis techniques.

In this approach we use information from the release history with respect to different keywords instead of feature data. This information was reflected onto the module structure of the source code and visualized to generate the high-level views of a software system. Independent from our research work Yamamoto et al. investigated variants of the BSD system for similarities as well [144]. They mainly use *CCFinder* by Kamiya et al. [84] to compute similarity metrics of the source code. In contrast to our work, their aim

lies on the overall similarities between different products, rather than the type, amount and distribution of information flow between the variants.

Chapter 5

Case Studies

To demonstrate and evaluate our approach we applied the steps of our *EvoZilla* approach to the *Mozilla Application Suite*. It is a large scale open source software project. The source code, the release history data, and the problem report data are freely available on the *Mozilla* developers web-site. Primary objective of this case study is to point out evolutionary hot-spots and their structural properties with respect to their evolution.

5.1 About Mozilla

The *Mozilla Application Suite* forms large software development tool that is a blend of XML document processing, scripting languages, and software objects. It is used to create interactive, user-focused applications. The architecture of *Mozilla* consists of the base system mainly written in C/C++ and—on a higher abstract level—a number of scripts and user interface definitions tied to the base system. The sum of both sub-systems represent the actual application. It can be understood as graphical web-shell interpreting a large complex script building an Internet browser.

For selected releases we extracted some metrics to describe to *Mozilla* source code. Table 5.1 lists some of the releases together with the number of source code files (*.h, *.cpp, and *.c) and the total lines of C/C++ source code (LOC) per release. The time interval between each two subsequent releases is about half a year. The table shows that the amount of source code is increasing from release to release. For instance, the number of source files increased by 360 files or 606,509 LOCs from *Mozilla* release 0.92 to 1.7. An interesting peak in terms of number of files and lines of code is by release 1.4 with 11,585 source files and 3,966,466 LOCs. Up to this release source code has been added permanently due to addition of new features or extension of existing features. Then, from release 1.4 to 1.6 the amount of source code decreased by 405 source code files. In particular, several *.c and *.cpp files have been removed or sourced out to libraries. In the next release the source code again increased.

Thus, we can expect an interesting case study with a number of changes in the source code during the first half of the project. In the subsequent time-interval a number of re-engineering events might be possible

Table 5.1 Mozilla releases with the number of files (NOF) and lines of code (LOC) metrics.

Release	#.h	#.cpp	#.c	NOF	LOC
0.92	4,695	3,847	1,600	10,142	3,306,122
0.97	4,824	3,896	1,635	10,355	3,518,124
1.0	5,258	3,961	1,970	11,189	3,868,025
1.3a	5,464	4,119	1,806	11,389	3,924,064
1.4	5,585	4,168	1,832	11,585	3,986,466
1.6	5,473	4,161	1,546	11,180	3,835,173
1.7	5,662	4,278	1,562	11,502	3,912,631

to detect. Before we can start with the analysis, we need to prepare the *Release History Database*.

5.2 Modification reports

In this section we evaluate our *Release History Database* against the data from the *Mozilla Application Suite*. Since the *Mozilla* project got a new focus in 2003 by establishing the *Mozilla* foundation, their lead project at the time of writing is called *Firefox*. Therefore we decided to stick with the *Mozilla Application Suite* and to update the *Release History Database* in 2005 with new modification- and problem reports. As of the beginning of 2004, *Mozilla* can be characterized as large scale software system consisting of more than 90 modules distributed over 2,500 directories, populated with more then 30,000 files and comprising more than two million lines of C/C++ code. This is the result of more than 6 years active and ongoing development.

From CVS and *BugZilla* we imported more than 490,000 and more than 250,000 records, respectively. Additionally, more than 60,000 source code changes were downloaded and integrated into the *Release History Database* to track the evolution of source code dependencies.

5.2.1 Evaluation of the Release History Database

Listing 5.1 File change history of selected files with number of modification reports and problem reports (excerpt).

```
security/manager/pki/src/nsPKIModule.cpp
.  1  3  1  1  1  .  .  .  .  .  .  .  1  .  1  .  .  .  .  .  2  .  .   # of MRs
.  .  .  .  1  .  .  .  .  .  .  .  .  1  .  7  .  .  .  .  .  2  .  .   # of PRs

security/manager/pki/src/nsNSSDialogs.cpp
.  2 11  9  5  .  2  1 11  1  1  3  1  .  .  1  .  .  .  .  3  3  .  1   # of MRs
.  1  1  2  4  .  2  1  9  1  2  3  1  .  .  1  .  .  .  .  2  3  .  1   # of PRs
```

In this section we evaluate our approach according to import, time-scale, historical, and coupling aspects of *Mozilla*. The results are based on data available per December 14th, 2002.

At that time 36,662 artifacts and 433,833 modification reports were imported to the **cvsitem** and **cvsitemlog** tables, respectively. From these artifacts, 23,540 were identified to have a bug report ID in one of their associated modification reports. In total 158,491 references to bug reports were found which resulted in a final number of 28,456 bug reports imported to the *Release History Database*. Thus, out of the total number of 180,000 bug reports stored in *BugZilla*, we filtered a solid sample of roughly a sixth of the full set. This sample exhibits an important characteristic for evolution analysis: they are referenced by modification reports and can be linked to certain changes in particular files or logically-coupled files indicating what was changed, what the result of the change was, and when the change happened.

5.2.1.1 Time scale

For a first analysis of evolutionary aspects, e.g., system growth or change rate, it is necessary to create a time-scale based on an appropriate granularity [87]. In our semi-automatic approach we used the symbolic names retrieved from the CVS log files, e.g., *MOZILLA_1_0_RELEASE*, as indicators. During the import process each occurrence of a symbolic name is counted and the total number together with the most actual date of a modification report are stored in the *Release History Database* (see **cvsalias**). The counting process considers symbolic names associated with the main trunk only, since they indicate the affiliation to the core architecture. These values are then used as indicators for possible release dates. A Java program selects entries by using a regular expression with groups to priorize the results, e.g., (MOZILLA.*RELEASE)*(.*BASE)*(.*RELEASE)*(.*)* , whereas only candidates with a high number of “votes” are selected. Since the *Mozilla* project team has published new releases on a nearly monthly interval, we also used a monthly interval for our further considerations. The results for our the reconstructed release dates are listed in Chapter C.

Listing 5.2 Problem report history.

```
+------+--------+----------------------------------------------------------------------+
| bugid|severity|short_desc                                                            |
+------+--------+----------------------------------------------------------------------+
|169943|blocker |Form submit buttons not working [embedding apps]                      |
|110155|critical|Convert all arbitrary content <tree>s to <listbox>                    |
| 97044|critical|PSM is passing null string to preferences @nsPrefBranch::QueryObserver|
| 92475|critical|Need error msg for expired CRLs.                                      |
| 70595|major   |Need to make nsIPrompt accessible to nsIChannelSecurityInfo object    |
|169932|normal  |Replace wstring with AString in IDL                                   |
| 97667|normal  |nsIInterfaceRequestor.idl needs freezing                              |
| 81257|normal  |Accepting Root CA- View and Policy buttons not working.               |
| 79153|normal  |No indication that a key is being generated.                          |
| 78012|normal  |Finish up the Certificate Viewer for PSM 2                            |
| 74803|normal  |Should make global data const where possible                          |
| 74436|normal  |PSM 2.0 needs to use WindowWatcher service instead of AppShell        |
|131393|enhance |outliner content view should support icons from src attribute         |
| 44042|enhance |Wording on security-alert dialog is confusing                         |
| 31896|enhance |lock icon distinguish between weak and strong encryption              |
|......|........|......................................................................|
```

5.2.1.2 Release history

Based on the time-scale we can compute the *release history* for all artifacts in the *Release History Database*. An artifact is marked when it first appears and the mark is updated every time the artifact is modified in one of the time-slots. For example, the file nsPKIModule.cpp from 5.1 has been introduced in release 33, modified in releases 34 trough 37, then again in 45 and 47, and finally in 53. This leads to the following *release sequence number*: $<33,34,35,36,37,45,47,53>$. These sequence numbers are recorded and used in the detection of logical coupling [62]. Another aspect of the release history are the number of modifications and problem reports associated with every artifact and time-slot. The two example files, nsPKIModule.cpp (109 lines) and nsNSSDialogs.cpp (747 lines), in 5.1 were introduced in release 33 (*2001-02-10, MOZILLA_0_8_2001020916_BASE*) and remained in the main trunk until the latest release (*2002-12-02, MOZILLA_1_2_1_RELEASE*). Although the first file has been modified less frequently and also has lesser problem reports, the $source - line/bugreport$ ratio is better for the second file (9.9 compared to 21), which means that the code of the first file is more error-prone. To retrieve more detailed bug report related information a simple SQL statement can be used, e.g., to list all bug reports for nsNSSDialogs.cpp:

Listing 5.3 Query for problem reports of file nsNSSDialogs.cpp.

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

Besides a number of "normal" rated bug fixes (not all are shown in 5.2), one blocking problem (blocks development and / or testing work), two critical problems (crashes, loss of data, severe memory leak), one major problem (loss of function), and two requests for enhancement were assigned to this file.

5.2.1.3 System history

From the release history of every artifact the release history of the over-all system can be derived.

Figure 5.1 Spectrogram of changed files in Mozilla.

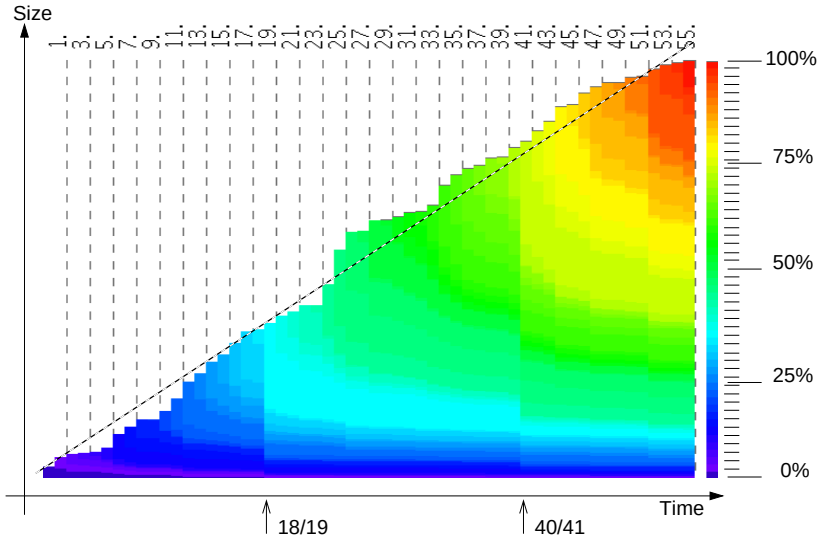


Figure 5.1 depicts 56 releases of *Mozilla* as *system history* view, which shows an approximately linear growing of the system. The rightmost bar is used for scaling and represents 100% or 34,847 artifacts. Label "0" has been assigned to the leftmost release. Due to space limitations only releases with odd numbers are labeled, and dashed lines indicate the boundary between an odd and an even labeled release. The coloring in Figure 5.1 indicates that about 50% of the files have been modified within the last quarter of project duration, even though only about 25% were introduced in this last period. In [64] this approach has been applied on subsystem-level to compare growing, change rate and stability. Applied on system-level, it allows to compare the evolution of different systems on a very high level, e.g., to compare code maturity.

5.2.1.4 Coupling

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

For instance, we selected all 33 bug reports of nsNSSDialogs.cpp and evaluated the number of artifacts referenced by these reports. We found 456 different artifacts which were affected by the selected bug reports. The topmost referenced files were nsNSSCertificate.cpp with 16 references, nsNSSComponent.cpp with 13 references, and nsINSSDialogs.idl with 11 references. Not surprising, they all belong to the same sub-module security/manager/pki. The first file from a different sub-module was nsPKCS12Blob.cpp from security/manager/pki with 7 references. We also found a single problem report (see [6] for report 88,413) with references to 373 different files. A change in an interface of a base class required the modification of this large number of files. These relations between bugs can be used to build groups of reports which refer to similar problems.

Table 5.2 Problem reports of BugZilla.

\ Status Resolution	assigned all / ref	closed all / ref	reopend all / ref	resolved all / ref
undefined	7,016 / 510	0 / 0	988 / 173	0 / 0
duplicate	0 / 0	282 / 5	0 / 0	13,855 / 66
fixed	0 / 0	333 / 63	0 / 0	14,806 / **7,705**
invalid	0 / 0	315 / 6	0 / 0	4,551 / 43
later	0 / 0	0 / 0	0 / 0	4 / 4
moved	0 / 0	0 / 0	0 / 0	24 / 0
remind	0 / 0	3 / 0	0 / 0	1 / 1
won't fix	0 / 0	92 / 1	0 / 0	1,823 / 46
works for me	0 / 0	359 / 2	0 / 0	9,765 / 93

\ Status Resolution	new all / ref	unconfirmed all / ref	undefined all / ref	verified all / ref
undefined	19,582 / 415	3,348 / 16	1,318 / 199	0 / 0
duplicate	0 / 0	0 / 0	0 / 0	41,648 / 487
fixed	0 / 0	0 / 0	0 / 0	36,620 / **18,940**
invalid	0 / 0	0 / 0	0 / 0	9,116 / 113
later	0 / 0	0 / 0	0 / 0	6 / 5
moved	0 / 0	0 / 0	0 / 0	73 / 1
remind	0 / 0	0 / 0	0 / 0	5 / 5
won't fix	0 / 0	0 / 0	0 / 0	3,123 / 69
works for me	0 / 0	0 / 0	0 / 0	15,569 / 308

5.3 Problem reports

This section describes mainly the quantitative results of the import process. Objective is to gain an insight into the reliableness of the extraction of problem report IDs from the modification reports. In the resulting data-set we expect to find only some false positives, i.e., rejected problems or IDs which do not point to a valid problem report.

5.3.1 Distribution of problem reports

The first test relates relates to the distribution of the identified problem reports with respect to the different different classifications which are support by the problem tracking system. An important classification is the *fixed* resolutions since it indicates that a reported problem has been identified as such and has been solved as well.

Table 5.2 indicates that 91% of the referenced reports *ref* fall either into the group *fixed/resolved* (7,705) or *fixed/verified* (18,940). The other categories are sparsely filled which may indicate a positive false detection or incorrect tracking status of problem reports. If we compare this data with all reports downloaded from the *BugZilla* database, we recognize that a large number of problem reports 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 problem reports 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 modification reports the chances to pick a false or correct PR ID would be equal.

We now give quantitative results of the link validation method applied on the reconstructed links between modification and problem reports. In total 33,499 links have been reconstructed. Our link validation method checked all these links and according the regular expression rated 27,835 links as *high,* 4,908 links as *medium,* and 758 links as *low*. Investigating the patch information 9,379 of links rated *high* were vali-

Table 5.3 Products and problem reports.

	fixed reports			all reports		
Product	all	found	%	all	found	%
Browser	35,520	20,396	57.42	129,889	22,208	17.10
Browser (locale)	2	2	100.00	2	2	100.00
BugZilla	1,630	9	0.55	4,564	26	0.57
CCK	459	7	1.53	688	11	1.60
Calendar	371	192	51.75	709	197	27.79
Camino	535	1	0.19	2,249	1	0.04
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
Grendel	9	0	0.00	41	0	0.00
JSS	82	4	4.88	122	4	3.28
MailNews	7,089	4,583	64.65	29,112	4,978	17.10
Mozilla (locale)	234	0	0.00	473	1	0.21
MozillaClassic	178	6	3.37	479	10	2.09
NSPR	445	371	83.37	713	399	55.96
NSS	839	517	61.62	1,388	546	39.34
PSM	862	413	47.91	2,851	442	15.50
Phoenix	208	8	3.85	1,180	9	0.76
Rhino	115	0	0.00	179	1	0.56
Tech Evangelism	1,314	6	0.46	4,908	16	0.33
Webtools	239	1	0.42	617	2	0.32
mozilla.org	1,126	5	0.44	2,122	15	0.71

dated that is 33%. For the other two confidence categories we obtained similar values. 3,425 not validated references face 1,483 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.

5.3.2 Products and problem reports

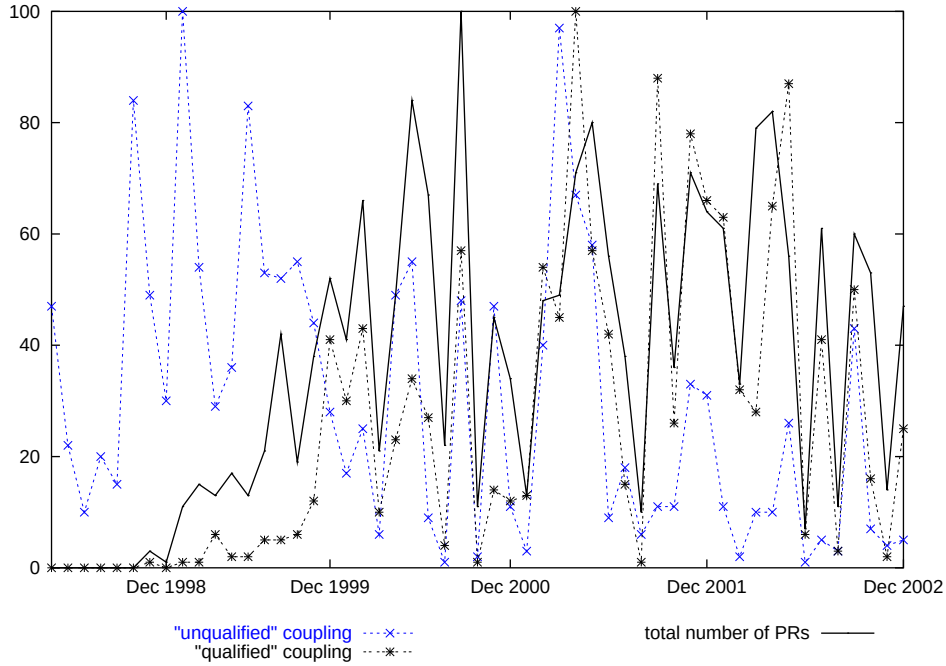
A summary overview about different products, the total number of problem reports and their validated linkage with modification reports is depicted in Table 5.3. Results for reports having status fixed are listed in the left column, whilst the right column labeled *all reports* lists the number of reports in the database regardless of their status. In the most important category *Browser* we found 20,396 of 35,520 downloaded reports which yields to a success rate of 57.42%. Most of the reports found are resolved as *fixed* and only a minor part belong the different resolutions such as *undefined* or *invalid*. Interesting are also other product 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.

5.3.3 Correlation with modification reports

In the following we will refer to every instance of a pairwise modification using the term *file-pair*. This is in contrast to *logical coupling* which denotes pairs of files commonly modified regardless of the modification rate.

From the total set of 36,661 files we found about 4.26Mio file-pairs, 2.55Mio of them were without associated PR – we call them *unqualified* coupled – and 1.89Mio file-pairs coupled via at least one PR. The latter type is called *qualified* since PR data provide information which allows further reasoning about

Figure 5.2 Problem reports and coupling.



modifications. 1,337 files had no relationship with any other file fulfilling the selection criterion.

The distribution of *unqualified* coupling (maximum 87,639), *qualified* coupling (81,782) and the distribution of problem reports (maximum 1,565) found for the whole *Mozilla* project are depicted in Figure 5.2. The values are normalized against their respective maximum values. Whereas for the first 10 releases virtually no couplings via problem reports are recorded, the number of *qualified* couplings increased and followed for about 2 years the graph of *unqualified* couplings. We assume that the bug-tracking system *BugZilla* was not available for the first releases of the *Mozilla Application Suite.*

From the run of the curves we deduce a strong dependency between problem reports and *qualified* coupling. A χ^2-test we applied on the number of *qualified* and *unqualified* couplings did not support the assumption that these types of coupling have any dependencies. Revealing such dependencies could be beneficial for prediction models.

As an example for the distribution of the modification rate, we depict the their number for the year 2002 in Figure 5.4. The first column indicates the release dates whereas the consecutive columns indicate

Table 5.4 Distribution of file-pair modification rate in 2002.

	Rate															
Date	1	2	3	4	5	6	7	8	9	10	11	12	13	14	16	17
01.23	58,436	2,321	305	68	20	6	4	1	1							
02.05	27,286	433	56	12	5											
03.11	25,968	2,748	492	106	39	16	9	4	2	2	0	0	0	1	1	
04.22	53,138	3,274	1,309	529	880	125	63	46	14	8	4	3	2	0	0	1
05.31	40,925	39,539	7,272	710	431	60	3	0	1	1	2					
06.06	5,655	87	2													
07.23	31,560	2,882	1,577	176	29	15	4	6	1	1	1	1				
08.05	3,768	481	84	19	2											
09.12	63,867	8,278	2,376	421	92	42	7	33	9	8	2	1				
10.17	17,462	1,233	262	67	20	10	1	0	1							
11.05	4,843	154	12	3	1											
12.12	22,750	1,324	663	63	43	36	2	2								

the modification rate of file-pairs. For instance, the highest rate was in April where one file-pair has been modified 17 times. Common to all 5 years of our observation period is the increase in the modification rate during spring and autumn whereas the total number of file-pairs not necessarily increased. E.g., February and March have about the same number of file-pairs but the modification rate for certain file-pairs is much higher. This suggest that periods with higher programming/bug-fixing activities are during none-holiday seasons such as March or September.

5.3.4 Coupling distribution

To get a better overview about the coupling between the two modules we created a high-level view of the structure of *content* – in lower left area – and *layout* – in the upper right area. In Figure 5.3 each graph summarizes the inter- and intra-coupling over a one year period except for graph 5.3(a) in which two years are depicted. To obtain a more compact representation we shifted data from sub-directories containing less than 125 entries to the next higher level until sufficient data has been aggregated. The colored edges between nodes indicated the type of logical coupling:

- green edges indicate that no problem reports for the majority of logically-coupled files were found;
- red edges indicate that for a majority of logically-coupled files problem reports were found; and
- blue edges indicate that the number is about equal (we refer to this edge type as *unbiased*).

Since virtually no problem reports were recorded for the first two years, the type of logical coupling is clearly biased towards *unqualified* coupling. As mentioned in our third experiment, the sub-module, i.e., directory, *content/xul* has been added later to the project. This can be the reason for the large number of *qualified* edges connecting *content/xul* with other nodes in graph (see Figure 5.3(a)).

If we compare the 4 graphs from the top/left to the bottom/right we recognize that the type of logical coupling changes from *unqualified*, over *unbiased* to *qualified*. This trend is also supported by the distribution of problem reports and logical coupling depicted in Figure 5.2. There the number of *unqualified* coupling drops in the middle of 2001 (see Figure 5.3(c)) to a very low rate, whereas the number *remains* on a high level until the end of 2002 (see Figure 5.3(d)).

5.4 Feature analysis

Feature analysis builds the foundation for a detailed structural analysis. The resulting file-sets of this analysis are the input to our *EvoGraph* approach in the subsequent section. Following, the first step in feature analysis is the determination of an appropriate mapping of the abstract concept of features onto concrete realizations.

5.4.1 Feature extraction

5.4.1.1 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 5.5).

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.

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 *http://bugs.gnu.org/cgi-bin/gnatsweb.pl*), in the GNU glibc library, originally reported by *http://mail.gnu.org/archive/html/bug-glibc/2001-07/msg00130.html* 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

Figure 5.3 Pair change coupling between content and layout.

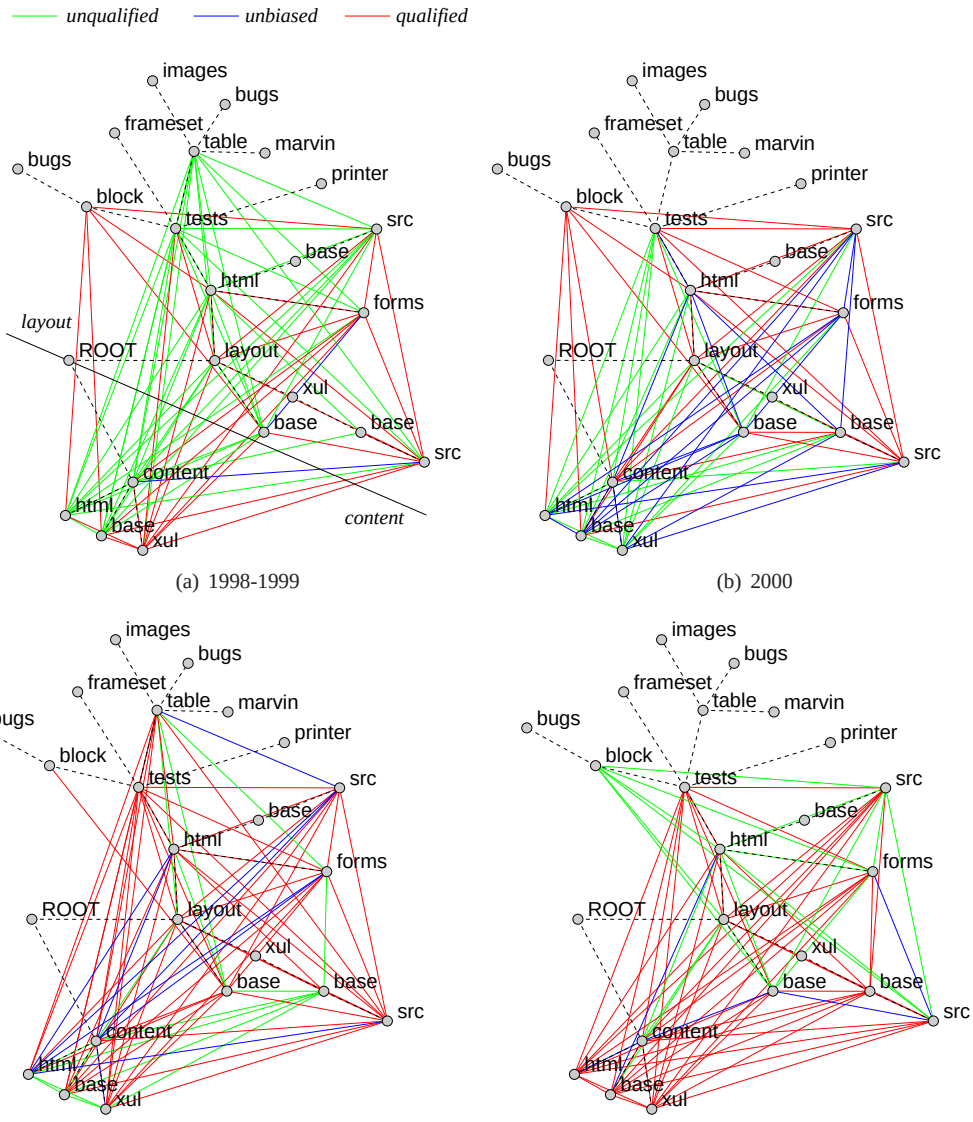


Table 5.5 Mozilla and OS version.

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

Table 5.6 Scenario definitions and features.

Scenario	Description	Feature	Fill style	ID	Files
sNull	Mozilla start / blank window / stop	fCore		00	705
sTC-HTTP	TrustCenter.de via HTTP	fHttp		01	28
sTC-HTTPS	TrustCenter.de via SSL/HTTP	fHttps		02	6
sTC-File	read TrustCenter.de from file	-		-	-
sXML	XML Base	fXml		09	65
sMathML	mathematics in Web pages	fMathML		08	13
sAbout	"about:" protocol	fAbout		10	3
sfBlank	read blank html page	fHtml		03	76
shBlank	blank html page via HTTP	-		-	-
-	-	fImage		04	3
sChromeGIF	chrome://global/content/logo.gif	fImageGIF		07	4
sPNG	image: Portable Network Graphics	fImagePNG		05	10
sJPG	image: Joint Photographic Experts Group	fImageJPG		06	16

reason why the statically linked versions of *Mozilla* produced only profile data when running on *RedHat 8.0* (see 5.5) 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 [113]. 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.

5.4.1.2 Feature extraction (scenario definition)

The 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 applied the software reconnaissance technique [139, 140] within our *Linux* (*RedHat 8.0* & *SuSE 8.1*) development environment. GNU tools [8] have been used successfully in [47] to extract feature data.

First, we created a single statically linked version of *Mozilla* with profiling support enabled. From several test-runs where the defined scenarios (see Table 5.6) were executed, we created the call graph information using the GNU profiler. The call graph information was 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 a higher level of abstraction. In the next step, "feature data" were extracted from file name mappings using set operations. For example, the fXml feature was extracted by the following expression (see also Figure 5.4):

$$\mathrm{fXml}(\mathrm{sMathML} \cap \mathrm{sXML}) \setminus (\mathrm{sNull} \cup \mathrm{sTC\text{-}HTTP} \cup \mathrm{sPNG} \cup \mathrm{sfBlank} \cup \mathrm{shBlank} \cup \mathrm{sChromeGIF})$$

The names in the above expression represent the set of files extracted in the previous steps from the executed scenarios. Table 5.6 also lists the names assigned to the features, the fill styles which are used for their

Table 5.7 Total number of logical couplings between features.

	Feature										Period				
Feature	01	02	03	04	05	06	07	08	09	10	<2000	2000	2001	2002	Total
fHttp	0	20	10	7	0	2	2	0	7	4	24	63	155	129	371
fHttps		0	2	1	0	0	0	0	2	0	0	1	69	61	131
fHtml			0	6	0	1	46	16	29	1	87	116	140	122	465
fImage				0	0	15	5	0	3	2	5	5	54	31	95
fImagePNG					0	0	0	0	0	0	0	0	3	5	8
fImageJPG						0	0	0	0	0	1	0	22	11	34
fImageGIF							0	17	36	0	15	61	75	38	189
fMathML								0	67	0	2	9	32	72	115
fXml									0	1	7	34	105	110	256
fAbout										0	2	1	2	1	6

visualization, and the number of files retrieved. Finally, we imported the filename-information into the *Release History Database* along with the release number of the program from which the data were retrieved. In our case it was *Mozilla* version *1.3a* with the official freeze date 2002-12-10 (even though we found one modification report with a time-stamp 2002-12-12).

5.4.1.3 Qualification and selection of problem reports

The distribution for all types of reports referenced from modification reports for the fCore feature (represented as white bars in the background) and the other extracted features (represented by different fill styles) is depicted in Figure 5.5 using a bi-monthly time-scale. Since the number of reports for the *core* is a magnitude larger than the number of reports for the other features, we used a scale ratio of $10:1$ for the boxes. The largest number of reports found for a single period was 3,628 ending on 2000-06-05. To visualize the fCore-to-feature ratio, the bottom of the white bars are shaded according to the ratio calculated.

It can be seen from the number of fixed problem reports for the features, that periods with less activities are during the summer time and at the end of each year. Thus we decided to use one year as time-frame in the visualization of feature dependencies in Figure 5.6.

5.4.2 Visualizing feature evolution

Visualization is a useful technique to present complex feature interrelationships. We use two different types of views to facilitate the understanding of evolutionary changes in large software systems:

- the *feature*-view focuses on the problem report based coupling between the selected features; and
- the *project*-view depicts the reflection of problem reports onto the directory structure of the project-tree.

5.4.2.1 Feature view: projecting problem reports onto features

This view focuses on the visualization of features and their dependencies via problem reports. The degree of coupling between two features is represented by edges whereas the number of references (i.e., the edge weight) from problem reports to files is expressed as line-width. Each line indicates the coupling of files through problem reports on the feature-level rather than the file-level. In fact, all entities contributing to a feature are drawn on the same position, which supports the impression that features are compared.

To reduce the amount of visible edges—one problem report can affect several items of a feature—and to visualize only important edges, we use the following criteria: (1) the edge weight between nodes is set to 200—if the actual weight is greater—to reduce the impact of outliers, i.e., dominant edges (upper bound); and (2) an edge must have at least 10% of the weight of the highest weighted edge to be visible (lower bound). As a consequence, every edge in Figures 5.6.b, 5.6.c and 5.6.d represent at least 20 references. Since the maximum number of references in Figure 5.6.a is 168, the smallest visible edges represent 17 references. To scale down from the large number of references we used the logarithm function to determine the visual line-width. The actual number of problem reports that features have in common and the total

$$\text{fCore} = (\text{sNull} \cap \text{sTC-HTTP} \cap \text{sTC-HTTPS} \cap \text{sTC-File} \cap \text{sMathML} \cap \text{sAbout} \cap \text{sPNG} \cap \text{sXML} \cap \text{sJPG} \cap \text{sfBlank} \cap \text{shBlank} \cap \text{sChromeGIF})$$

$$\text{fHttp} = (\text{sTC-HTTP} \cap \text{sMathML} \cap \text{sPNG} \cap \text{sXML} \cap \text{sJPG}) \setminus (\text{sNull} \cup \text{sTC-File} \cup \text{sAbout} \cup \text{sfBlank} \cup \text{sChromeGIF})$$

$$\text{fHttps} = (\text{sTC-HTTPS}) \setminus (\text{sNull} \cup \text{sTC-HTTP} \cup \text{sTC-File} \cup \text{sMathML} \cup \text{sAbout} \cup \text{sPNG} \cup \text{sXML} \cup \text{sJPG} \cup \text{sfBlank} \cup \text{shBlank} \cup \text{sChromeGIF})$$

$$\text{fHtml} = (\text{sTC-HTTP} \cap \text{sTC-HTTPS} \cap \text{sTC-File} \cap \text{sAbout} \cap \text{sfBlank} \cap \text{shBlank}) \setminus (\text{sNull})$$

$$\text{fImage} = (\text{sPNG} \cap \text{sJPG}) \setminus (\text{sNull} \cup \text{sTC-HTTP} \cup \text{sTC-HTTPS} \cup \text{sTC-File} \cup \text{sAbout} \cup \text{sfBlank} \cup \text{shBlank})$$

$$\text{fImagePNG} = (\text{sPNG}) \setminus (\text{sNull} \cup \text{sJPG} \cup \text{sfBlank} \cup \text{shBlank})$$

$$\text{fImageJPG} = (\text{sJPG}) \setminus (\text{sNull} \cup \text{sPNG} \cup \text{sfBlank} \cup \text{shBlank})$$

$$\text{fImageGIF} = (\text{sAbout} \cap \text{sChromeGIF} \cap \text{sTC-HTTP} \cap \text{sTC-HTTPS}) \setminus (\text{sNull} \cup \text{sfBlank} \cup \text{shBlank})$$

$$\text{fMathML} = (\text{sMathML}) \setminus (\text{sNull} \cup \text{sTC-HTTP} \cup \text{sTC-HTTPS} \cup \text{sTC-File} \cup \text{sAbout} \cup \text{sPNG} \cup \text{sXML} \cup \text{sJPG} \cup \text{sfBlank} \cup \text{shBlank} \cup \text{sChromeGIF})$$

$$\text{fXml} = (\text{sMathML} \cap \text{sXML}) \setminus (\text{sNull} \cup \text{sTC-HTTP} \cup \text{sPNG} \cup \text{sfBlank} \cup \text{shBlank} \cup \text{sChromeGIF})$$

$$\text{fAbout} = (\text{sAbout}) \setminus (\text{sNull} \cup \text{sTC-HTTP} \cup \text{sTC-HTTPS} \cup \text{sTC-File} \cup \text{sMathML} \cup \text{sPNG} \cup \text{sXML} \cup \text{sJPG} \cup \text{sfBlank} \cup \text{shBlank})$$

Figure 5.4 Set operations to extract feature data from the generated runtime information.

Figure 5.5 Distribution of problem reports fixed for core and selected features.

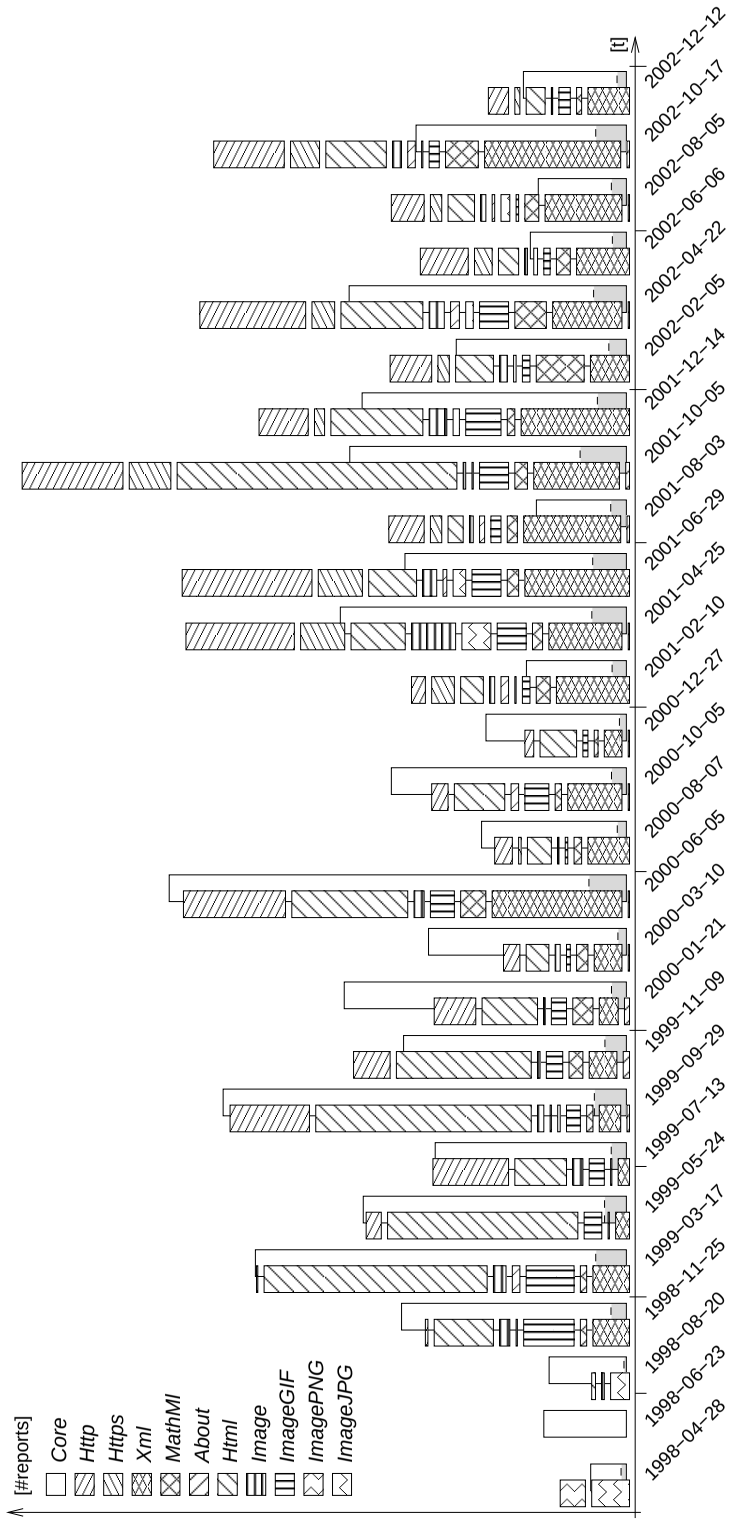
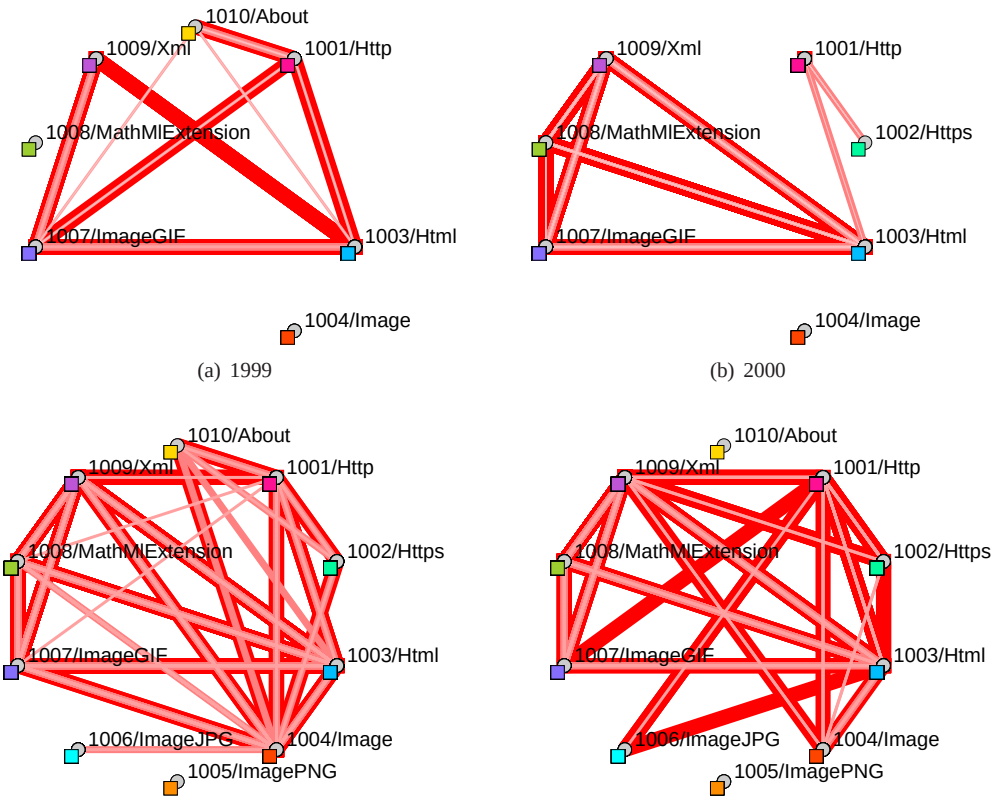


Figure 5.6 Dependencies between features reflected by large problem reports.



(a) 1999 (b) 2000

(c) 2001 (d) 2002

number of problem reports for every feature can be found in Table 5.7. Due to the small number of files—which means a small number of problem reports—we identified for the features fImageGIF, fImage, and fAbout and the threshold for references, almost no coupling is depicted even though some references do exist. If a feature is not shown in one of the figures, we could not find any problem report for the specified period or the threshold was not reached. An isolated feature indicates only "local" problem reports but no references to other features.

Figure 5.6 depicts the results for the observation periods 1999, 2000, 2001, and 2002, in which features are aligned on a circle and visualized according to the fill style given in Table 5.6. From 1998 till 1999—the result is depicted in Figure 5.6.a—we found virtually no coupling between features. The result changed in the subsequent observation periods not dramatically but constantly. In Figure 5.6.b the situation for 2000 is depicted and indicates that the focus shifted from *Http* to other features such as fHtml, fXml, and fMathML.

In 2001, the situation changed substantially and is depicted in Figure 5.6.c. The partially connected graph has turned into an almost fully connected graph and the number of reported and fixed problems have more than doubled (from 290 to 657). Except for feature fImagePNG, all features are affected by system-wide changes! In 2002 (see Figure 5.6.d) the situation slightly improved since the number of reported problems dropped to 580.

As a result, this kind of visualization can be used to point out phases of architectural deterioration on the basis of feature dependencies. Our window of observation for every figure was one year but that can be changed according to the given data set of the particular system under study. Such visualizations can be used (a) to assess the point of time when some restructuring or reengineering activities should be started and (b) to estimate the likely amount of resources to be required for changing particular features and/or

system parts.

5.4.2.2 Project view: generating the visualizations

Input data for *Xgvis* and *Xfig* used for visualization are generated by a Java program. This program accepts some arguments which allow the user to control the data selection and generation process (date-range, features, problem report severity). A critical step in the data generation process is the selection of parameters for the weights since this has a direct impact on the final layout. We used a ratio of $\geq 20:1$ for project-tree edges and problem report edges. This scheme gives more emphasis on the directory structure than to connections introduced by problem reports. In a first phase of the data generation process, the objects of the project tree are assigned to their respective nodes of the graph. The minimum child size (*minchild*) specifies which nodes remain expanded or will be collapsed. Collapsing means that objects from the sub-tree are moved up to the next higher level until the size criterion is met, but not higher than the first level below the "ROOT" node. Since the complete *Mozilla* project tree consists of more than 2,500 subdirectories, we had to simplify the resulting graph and so we cut off unreferenced directories. In a second step it is possible to move fewer referenced nodes to a higher level to obtain a more compact representation. The effect on the graph is that unreferenced leafs are suppressed, though they contain enough objects to meet the *minchild* criterion.

In Figures 5.7 and 5.8 the project directory tree is shown as gray nodes connected by black dashed lines. The root node is labeled "ROOT" and the features are indicated by filled in boxes according to the fill styles given in Table 5.6. Coupling between nodes as result of common problem reports are indicated by gray lines. Thicker lines and a darker coloring means that the number of problem reports that two nodes have in common is higher. Since the optimization algorithm tries to place connected nodes close to each other, stronger dependencies can be spotted easily.

One marginal problem is the limited layout area in the two dimensional solution space: all nodes must be placed at least somewhere within a single plane and the placement of nodes after the optimization is only indicative within a certain radius. Naturally, this radius depends on the total number of nodes. By zooming-in, it is possible to provide a better picture of otherwise overlaid areas. An n-dimensional solution space could yield better results but it is very difficult to visualize. In general, the layout after optimization is one possible solution. It also does not necessarily mean that a global minimum for the given distances has been achieved. *Xgvis* supports up to 12 dimensions, which would yield better results for the optimization step, if, for example, more features were used; but then the results are difficult to visualize.

5.4.2.3 Results: how features fHttp, fHttps, and fHtml relate

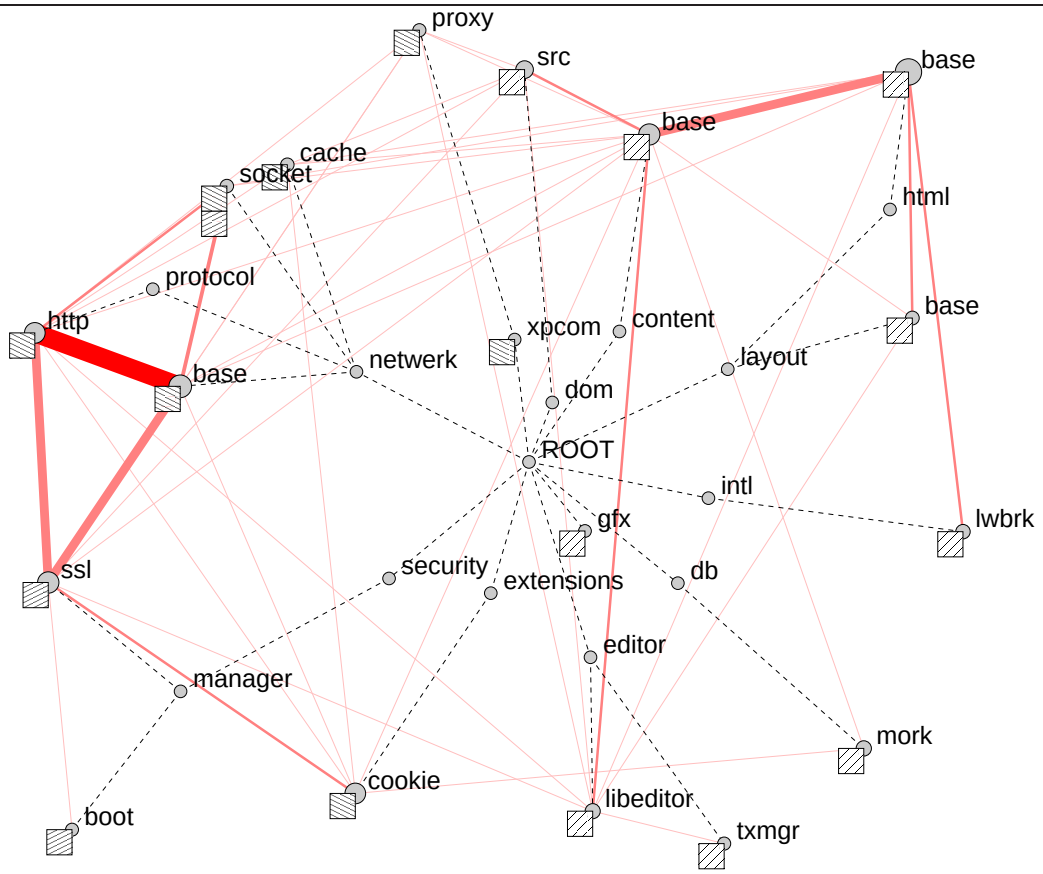
The three features fHttp, fHttps, and fHtml are depicted in Figure 5.7. As raw data we selected all problem reports for these features with the exception of problem reports classified as *enhancements* from the start of the project until the freeze date. Parameters, which influenced the project tree node selection process, were $minchild = 10$ (the number of artifacts, i.e. files in a subtree) and $compact = 1$ (the number of problem report referenced by a node).

For the optimization process, we weighted the edges of the project tree with 20 (this gives more emphasis on the project structure), whereas an edge introduced through a single problem report was weighted with 1. The factors $k = 0.2$ and $o = 0.2$ for the weight function were used to emphasize the spreading between nodes for visualization purposes.

The overall amount of problem reports detected for a node is indicated via the diameter of a node and features "hosted" by a node are attached as boxes. Easy to recognize is the placement of nodes belonging to the fHtml feature on the right side, and fHttp, fHttps on the opposite side of Figure 5.7. Interesting are the nodes *netwerk/base*, *netwerk/protocol/http*, and *security/manager/ssl* since they are coupled via 90 problem reports for *base – http* and 40 problem reports for each of the other two edges. This indicates a high degree of coupling between the features fHttp and fHttps.

Another interesting aspect is the spreading of the fHtml feature over 10 different nodes. Modifications may be hard to track since several files in different directories contribute to a single feature. Remarkable are the two nodes *content/base* and *layout/html/base*, since they are coupled via 35 problem reports although only 4 and 3 files are located in their respective directories.

Figure 5.7 Relation of features fHttp, fHttps, and fHtml via problem reports.



As a result, Figure 5.7 shows strong change dependencies for the involved features across different directories and points to architectural deterioration distilled from the evolution data:

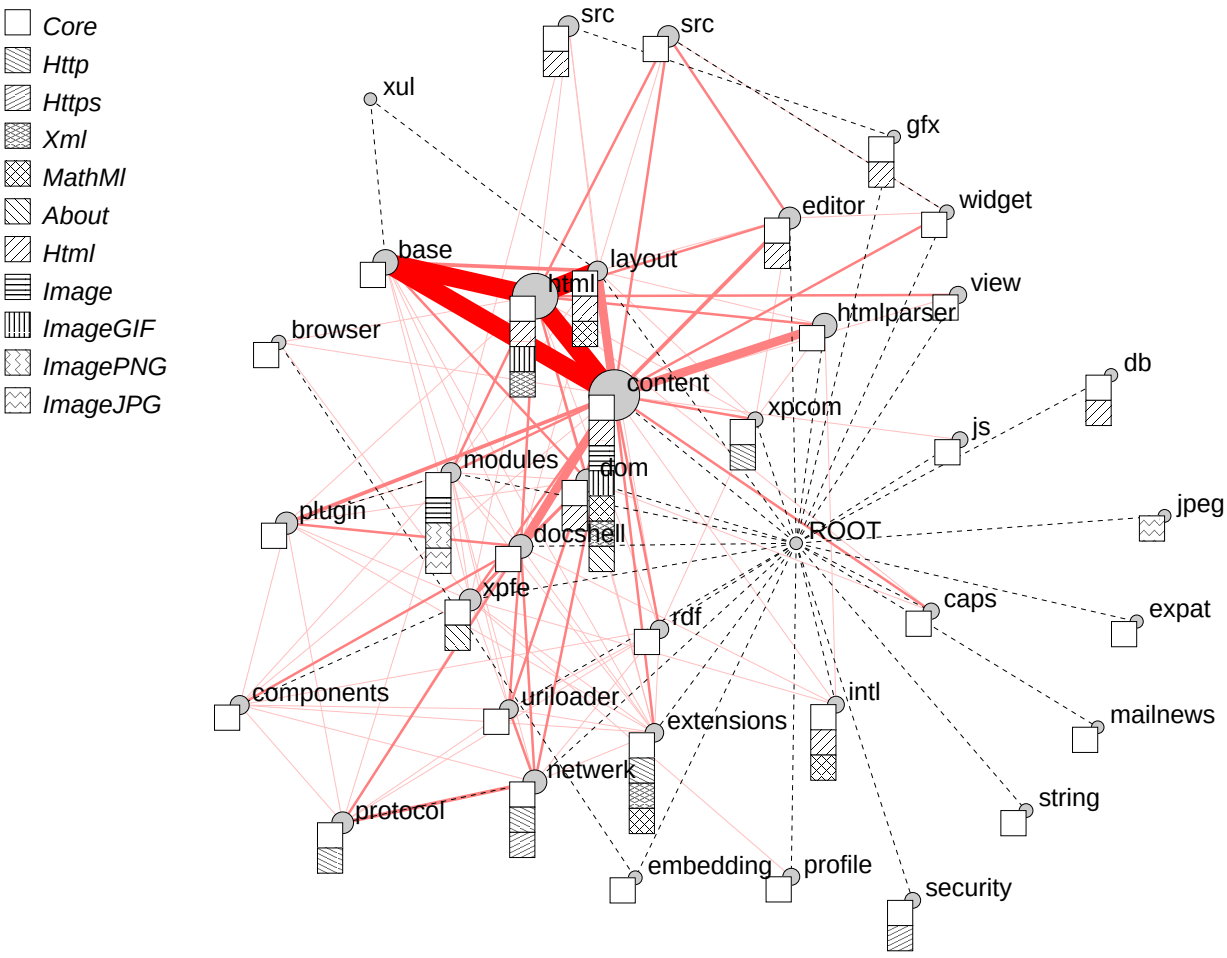
- dependencies across branches of the project tree may indicate that there exist dependencies such as invocation, access, inheritance, or association;
- frequently reported problems concerning certain source code locations of features can point to implementation or design problems;
- features spread across the project tree may indicate the involvement of large amounts of the code base for their realization and thus the impact of modifications can be difficult to predict.

5.4.2.4 Results: how all features relate with the core

The *core* feature and all other investigated features are depicted in Figure 5.8 on a coarse-grained level (edges that represent less than 5 references are omitted). For this configuration we selected all reports which were rated *major* or *critical*. We also set the minimum sub-tree size to 250 (*minchild*) entities and the minimum number of problem report references to 50 (*compact*). This resulted in a graph with 37 nodes and 315 edges induced by problem reports. By changing the values for *minchild* and *compact* it is possible to generate an arbitrarily detailed graph of the whole project. It is intuitive that the most critical subsystems in *Mozilla* are connected in the visualization, which is also supported by our findings.

The nodes with the highest density in *severe* problem reports are *content* (with 608 references), *layout/html* (444), *layout/xul/base* (223), and *layout* (212). Another interesting aspect is the spreading of edges. In total 343 connections between nodes are depicted. If we select only those edges representing ≥ 10 references, then we can find the following ranking: node *content* with 19 edges, *layout/html* 10, *docshell* 9,

Figure 5.8 Relation of core and features via problem reports.



dom 5, *layout/xul/base*, *netwerk* and *uriloader* with 4 edges each. The other nodes have 3 or less edges with such a weight. In total 23 nodes share edges with other nodes with ≥ 10 references but 19 of them are connected with *content*. This confirms the exceptional position of *content* which is also indicated by the 6 different features located there. As a result, pictures such as Figure 5.8 allow an analyst to draw conclusions as follows:

- nodes with frequent changes appear larger than others and can be spotted easily;
- unstable parts of a system such as *content* are located near the center of the graph and they are highly coupled with other nodes;
- features which have a common code base are attached to particular nodes and placed close to each other (e.g., $\mathrm{fMathML}$ and fXml); and
- specific feature sets (e.g., the fImage feature) that are scattered over several nodes can be easily spotted (e.g., nodes *jpeg*, *modules*, *layout/html*, *content*).

As a consequence, locations of intensive change history and scattering of features point to software parts that should be considered for further investigation in terms of eliminating high complexity or architectural deterioration.

After identification of such hot spots, architectural analysis tools [44, 51, 86] can be applied for a directed search of those code elements which are responsible for logical coupling. As example, we inspect frequently modified file pairs via manual analysis of modification reports, source code deltas, and call graph information. Table 5.8 lists some of the topmost modified file-pairs of the two directories *content*

Table 5.8 Frequently modified file pairs (subset).

P	M	Directory *content*	Directory *layout*
50	27	*events/src/nsEventStateManager.cpp*	*html/base/src/nsPresShell.cpp*
49	32	*xul/document/src/nsXULDocument.cpp*	*html/base/src/nsPresShell.cpp*
46	11	*base/src/nsDocumentViewer.cpp*	*html/base/src/nsPresShell.cpp*
38	21	*xul/document/src/nsXULDocument.cpp*	*html/style/src/nsCSSFrameConstructor/cpp*
25	12	*xul/document/src/nsXULDocument.h*	*html/base/src/nsPresShell.cpp*
23	13	*xul/content/src/nsXULElement.cpp*	*html/base/src/nsPresShell.cpp*
21	8	*xul/content/src/nsXULElement.h*	*html/style/src/nsCSSFrameConstructor.cpp*

and *layout*. Column "P" lists the number of modification reports with associated problem reports and column "M" lists the number of modification reports without associated problem reports. As the table indicates we can expect to find a strong relationship between event management (key, mouse, focus, etc.) located in directory *event*, the user interface components in *xul* (*XML User Interface Language*), and the visualization of HTML-content in *html*.

First, we show the dependency between *nsEventStateManager.cpp* (5,384 source lines) and *nsPresShell.cpp* (7,961 source lines) as listed in the first row of Table 5.8. Since both files were frequently and pair-wise modified, we can expect to find some evidence in one of the source code deltas from CVS. In revision *1.302* of *nsEventStateManager.cpp* a call to function *FindContentForShell()* has been added as new code segment. The corresponding delta of *nsPresShell.cpp* (revision 3.462) reveals that the function has been introduced with this revision. Thus, we have found a new call relationship between these two files.

Second, we investigated revision 1.122 of *nsXULElement.h* (673 source lines) and revision 1.766 of *nsCSSFrameConstructor.cpp* (14,330 source lines). An object type was modified; this modification—usually numerous files have to be modified if a type is changed—is also reflected in the number of affected files: 71 files in directory *content*, 23 in *layout*, 4 in *extensions*, and 2 in *xpfe*. From the source code deltas it is not possible to find a direct relationship, since only those source lines were modified where the data type was used in a declaration. An inspection of the call graph information from the feature extraction process reveals that the method *AttributeChanged()* of *nsCSSFrameConstructor.cpp* calls *GetMappedAttributeImpact()* of *nsXULElement.cpp* (5,444 source lines). The include file *nsXULElement.h* declares this method which takes an argument of the modified type.

5.5 Structural analysis with EvoGraph

After studying the evolution and dependencies of features on a system-wide global level, we now apply the *EvoGraph* approach onto the previously used dataset to identify those entities which are the causes for the evolutionary dependencies. Furthermore, the evolution of the structural dependencies is tracked down to method- and variable-level. New to our *Release History Database* are therefore the source code changes which we had to download from the *Mozilla* CVS repository to supplement our data-set. For this analysis we use three arbitrary file-sets which we selected from the following stickiness-view (see Figure 5.9). Two additional file-sets demonstrate the approach on feature and cross-language level.

5.5.1 File selection and co-change visualization

The features determining our file-set basically cover the applications web-browsing functionality such as loading and displaying a web-page, XML and MathML content, navigation etc. We obtained about 1000 files with 80 of them having a stickiness, i.e., cross-cutting co-change transactions, of more than 30. Figure 5.9 depicts these files whereas the size of the squares indicates the number of detected co-changes. As already identified in earlier research as complex files [112, 114], the most interesting files from the evolutionary perspective *nsCSSFrameConstructor.cpp*, *nsPresShell.cpp*, *nsGlobalWindow.cpp* are re-appearing as the evolutionary hot-spots. In contrast to earlier research, here we focus on the evolutionary properties caused by structural dependencies. The three beforehand mentioned files are depicted as the largest squares

Figure 5.9 Stickiness view: Topmost coupled files with respect to co-changes traversing the root-node of the module tree. Larger squares indicate files with a high logical coupling with other files.

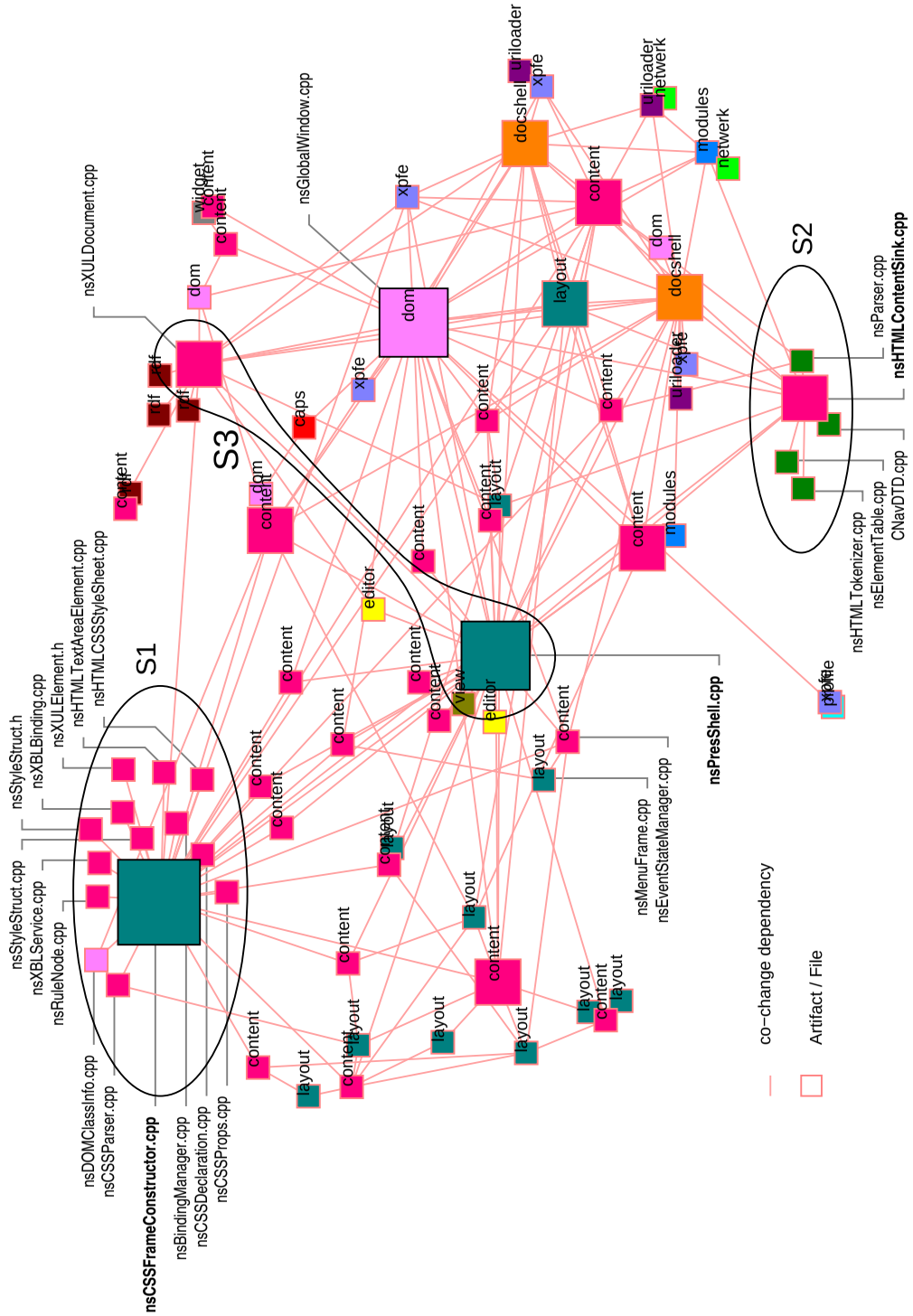
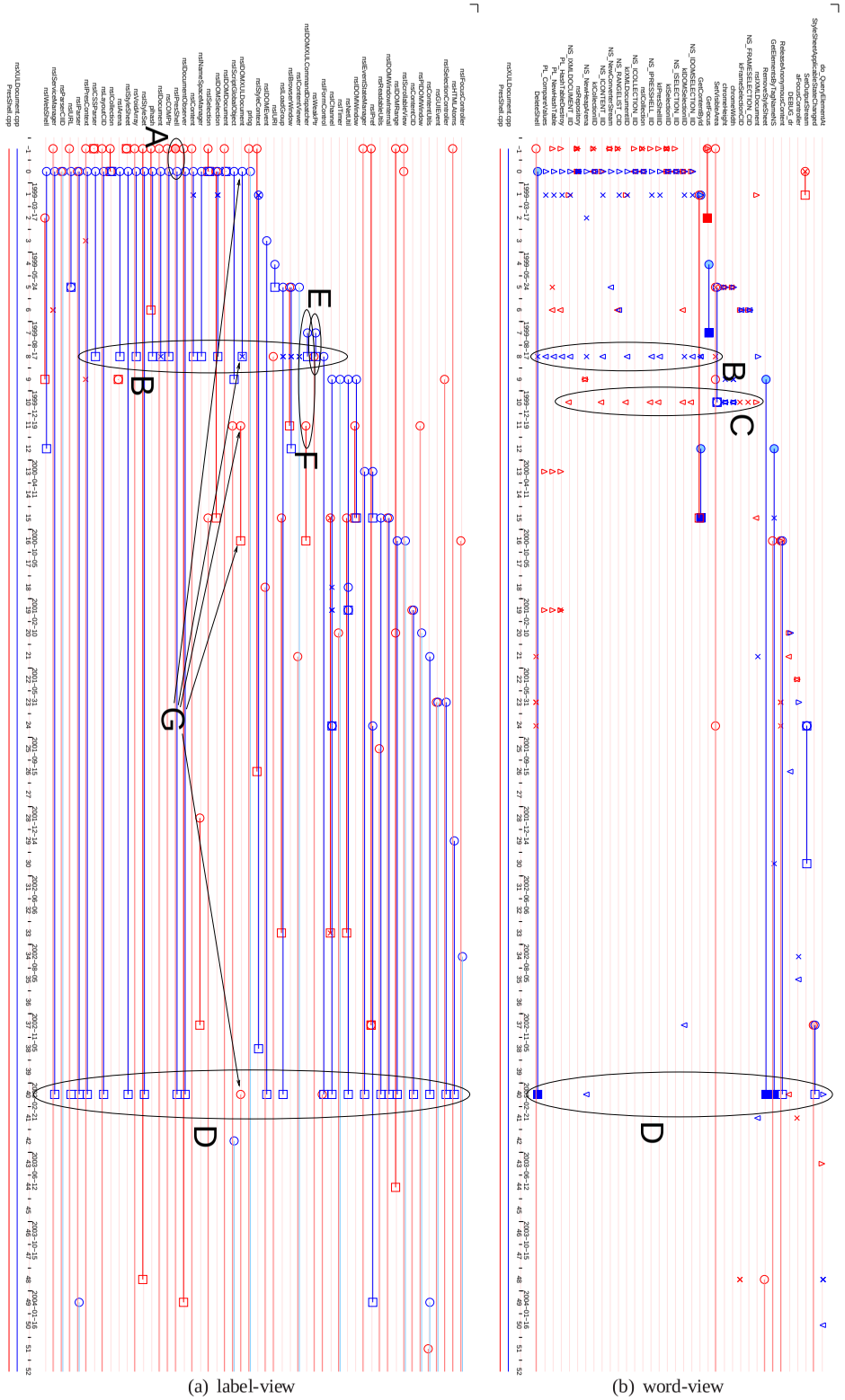


Figure 5.10 Commonalities between *nsXULDocument.cpp* and *nsPresShell.cpp*. The label-view depicted as graph (a) shows the *provides* and *depends-on* facts. Graph (b) is the word-view shows dependencies with respect to *uses* uses facts.



(a) label-view (b) word-view

indicating a high degree of stickiness. They are surrounded each by a number of other files represented as smaller squares sticking to the larger ones.

In the course of this case study we have also inspected the dependencies between the features fHttp and fHttps which build an additional set of files to the already 80 selected files.

5.5.2 Heuristics for Fact Extraction

Next, we provide an overview of the file types which are considered as information sources and the method how this information is retrieved from the selected files.

Interface definition files: a major element of *Mozilla*'s architecture is the component model called *Cross-Platform Component Object Model* (XPCOM). The publicly accessible interface definitions are stored in *.idl* files which can be parsed easily. From these files we can retrieve the following facts: (a) forward declarations of interfaces which are required by an interface. This is a *depends-on* relationship and defines an inheritance relationship or parameter type in a method signature; and (b) interface declarations which are recorded as *provides* fact.

JavaScript files: the JavaScript code is used to connect the GUI defined in *XML User Interface Language* (XUL) with the implementation code via component interfaces. XPCOM calls can be detected easily via regular expressions such as "`.*Components.interfaces\.(\w+).*`" whereas "`(\w+)`" matches the name of the interface.

C/C++ source code: due to the limited contextual information which is provided by source code deltas, the parsing of the C/C++ files must compensate this fact. One limitation is that relationship types (inheritance, method call, variable access, etc.) are currently not extracted. Only strings which represent identifiers in the program code are extracted. Since we focus on the structural impact of a change transaction rather than the exact meaning of a source code change we can omit such details. For our analysis we rely on information which can be extracted easily from the available deltas:

- comments are removed. Since some comment lines do not start with "`//`" or "`/*`" we use the following heuristics to remove possible comment lines: each line containing a sequence of word-characters and white spaces is filtered from the input stream. In our implementation we use the following regular expression: "`.*\w+\s+\w+\s+\w+.*`";
- a common practice to introduce new forward declarations is the application of *#include* pre-processor directive. Their detection is straight forward and the detected keywords are added to the *depends-on* vector;
- a new class definition is detected via the keyword *class*. The sequence following the keyword is recorded; and
- the remaining words are collected and used to identify possible commonalities or dependency relationships.

Currently, string matching is used to find corresponding relationships within vectors of extracted facts; namespaces are currently not supported. Furthermore, to overcome the limitations of an exact string matching approach, we utilize similarity metrics such as edit-distances (aka Levenshtein distance). This allows us the association of identifiers with file names if the naming was not consistent and minor naming variations have to be compensated.

5.5.3 Findings

We have found about 47,000 change entries belonging to 1,742 different change transaction for the selected observation period. Since we are interested only in those transactions which have an impact on the main trunk of the revision tree, we have to sort out those which affect branches only. This reduces the usable number of transactions by about 370 change transactions. The next section discusses the details about the extracted *adhesion* and *stickiness.*

Table 5.9 Excerpt of the top ranked logically-coupled files with respect to adhesion (A) and stickiness (S).

LC	Module	File	Module	File	#
A	*layout*	*nsBlockFrame.cpp*	*layout*	*nsBlockReflowState.cpp*	278
A	*mailnews*	*nsImapMailFolder.cpp*	*mailnews*	*nsImapProtocol.cpp*	226
A	*content*	*nsDocument.cpp*	*content*	*nsXULDocument.cpp*	212
...	...	...	...	...	...
A	*xpfe*	*navigator.js*	*xpfe*	*navigator.xul*	140
...	...	...	...	...	...
S	*content*	*nsEventStateManager.cpp*	*layout*	*nsPresShell.cpp*	97
S	*content*	*nsXULDocument.cpp*	*layout*	*nsPresShell.cpp*	87
S	*content*	*nsXULDocument.cpp*	*dom*	*nsGlobalWindow.cpp*	84
S	*content*	*nsCSSParser.cpp*	*layout*	*nsCSSFrameConstructor.cpp*	81
S	*content*	*nsGenericHTMLElement.cpp*	*layout*	*nsPresShell.cpp*	81
...	...	...	...	...	...
S	*content*	*nsEventStateManager.cpp*	*layout*	*nsMenuFrame.cpp*	35
...	...	...	...	...	...

5.5.3.1 Comparison of adhesion and stickiness

As already pointed out the *adhesion* should be higher than the *stickiness*. Table 5.9 depicts the conditions in the case study for arbitrary files—*.h* files are not considered here—with respect to their logical coupling. Some characteristics of files with a high adhesion is that they have also a certain commonality in their names indicating a structural relationship. Besides the large number of *.cpp* files, we found also a pair of user interface files which indicates the close structural relationship between function (*navigator.js*) and layout (*navigator.xul*). At the lower end with the lesser logical coupling are the relevant file-pairs stemming from different modules having a high stickiness. A threat to validity of the *Release History Database* are inconsistencies in the release history of the *Mozilla Application Suite* due to duplicated files. One file with a high stickiness is *nsChromeRegistry.cpp* which existed until release 1.7 in the modules *chrome* and *rdf* as well which leads to a false positive result. In our case study we have only actually used files since we exploit the run-time information to determine the respective file-sets.

5.5.3.2 Quantitative evaluation

Based on the results of step 4—mining of change transaction data—we analyze the tendency of added and removed *depends-on* facts depicted in the label-view. We use the two indicated file-sets S1 and S2 and an arbitrary pair of files with a high number of co-changes, in our case *nsXULDocument.cpp* and *PresShell.cpp* (in the following referred to as S3). Every set is divided into two subsets $_A$ and $_B$. The actual division of files is depicted in Figure 5.9. For every subset the individual number of added and removed dependencies are denoted as d_a and d_r, respectively. Similarly, their cumulated values are denoted as D_a and D_r, respectively.

To assess the structural stability we use two metrics. The first metric defined as

$$s = \frac{(d_a - d_r)}{(d_a + d_r)} \tag{5.1}$$

measures the survival rate of the aggregated structural changes. It yields a result in the range $[-1.. + 1]$. A negative result means a decreasing, a result around 0 would indicate periods of stagnation or without structural progress, and a positive result an increasing number of structural dependencies. The second metric is used to capture the quantitative aspect of the changes. We use

$$q = \frac{(d_a + d_r)}{(D_a + D_r)} \tag{5.2}$$

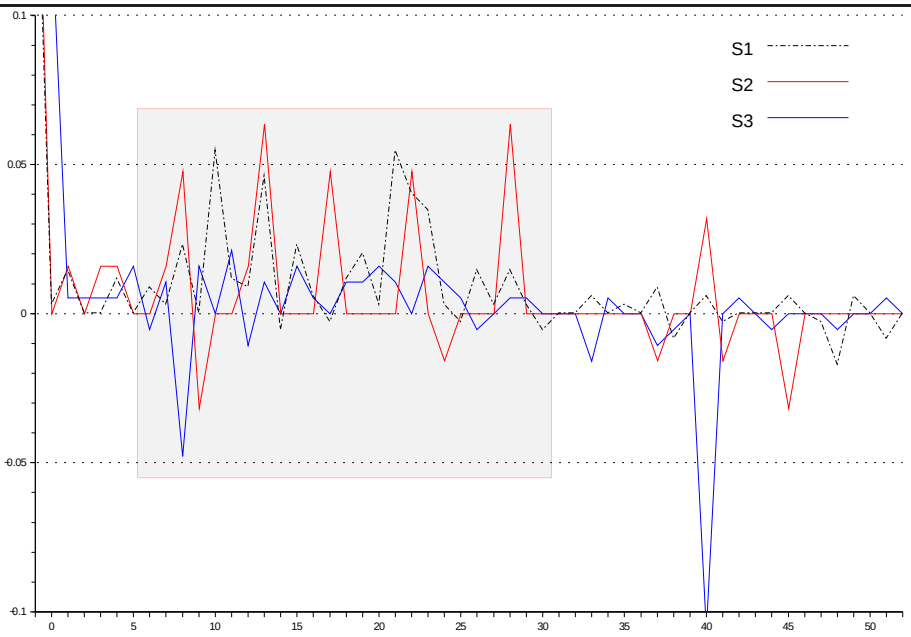
to estimate the fraction of structural changes for a given subset of files and assessment interval. This metric is in the range of $[0..1]$.

Since the overall time-scale is based on a monthly release interval, we divide the time-scale for this evaluation in the middle, which is after time-slot 25 (or release 0.9.6) yielding two *time-intervals* for assessment denoted as $1^{st}/2$ and $2^{nd}/2$ respectively (see Table 5.10). For interval $2^{nd}/2$ we expect that the development process stabilizes and therefore we should find lesser structural changes since the planned release 1.0 is only a few time-slots away.

$1^{st}/2$: interesting for the three sets S1, S2, and S3 is the significant difference in the survival rate of structural changes s_1. While for S1 we have a rate of more than 70%, it drops for $S2$ to about 50%, and is more divergent with 40% and 63% respectively for file-set S3. Considering the last two file-sets, 50% of the structural dependencies introduced were irrelevant and did not survive the first assessment interval. To make things even worse, about 75% (sum for both subsets) of the structural dependencies q_1 are introduced during this period. This implies that already in the middle of the project's lifetime time 38% of the structural changes were only temporary and had no permanent effect on the final product.

$2^{nd}/2$: In the second assessment interval, some stabilization was to expect which is also confirmed via the lower quantitative aspect of q_2 which is in the range of 8% to 18%. A surprising result for S3 is the negative value for s_2 with a high q_2 which means that a significant number of dependencies are removed again indicating a re-structuring event.

Figure 5.11 Comparison of the structural stability.



To compare the structural stability of the three file-sets, we compute the survival rate

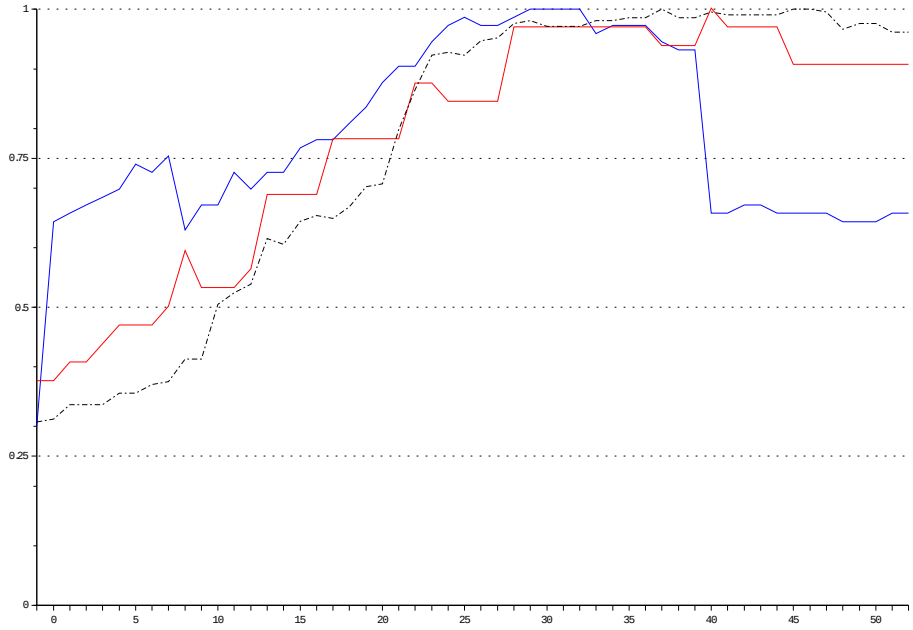
$$v_{t_s} = \frac{(d_a - d_r)}{(D_a + D_r)} \tag{5.3}$$

of the structural changes per time-slot t_s; The result depicted in Figure 5.11 indicates a turbulent development phase from time-slot 6 to 29 (shaded area). While for S2 and S3 some remarkable re-structuring events are detectable (negative values), there are virtually no such events for S1. Interesting to see is also an almost inverse correlation between S2 and S3. While structural elements are added to S2, S3 is stable or re-structured; and inversely, while S2 is stable or re-restructured, elements to S3 are added. This may indicate the existence of *daisy-chain* change patterns. Actually, in time-slot 40 we found the *nsIServiceManager* interface which has been removed from one file of S3 and added to S2 in the same time-slot.

Table 5.10 Structural survival rate of selected file pairs. Set determines the file-set, SS is the respective subset and I the assessment interval.

			word												label								
			U	provides				depends-on				uses				U	provides				depends-on			
Set	SS	I		d_{a_i}	d_{r_i}	s_i	q_i	d_{a_i}	d_{r_i}	s_i	q_i	d_{a_i}	d_{r_i}	s_i	q_i		d_{a_i}	d_{r_i}	s_i	q_i	d_{a_i}	d_{r_i}	s_i	q_i
S1	A	1	42	8	2	0.60	0.5	5	1	0.67	0.13	110	52	0.36	0.53	11	0	0	0.00	0.00	163	20	0.78	0.53
		2	18	4	0	1.00	0.2	5	1	0.67	0.13	30	9	0.54	0.13	0	5	1	1.00	1.00	33	31	0.03	0.18
	B	1	44	0	2	-1.00	0.1	16	9	0.28	0.54	63	17	0.58	0.26	2	0	0	0.00	0.00	57	8	0.75	0.19
		2	16	3	1	0.50	0.2	6	3	0.33	0.20	16	7	0.39	0.08	0	2	0	0.00	0.00	21	15	0.17	0.10
$\sum$			120	15	5	0.50	1.0	32	14	0.39	1.00	219	85	0.44	1.00	20	1	0	1.00	1.00	274	74	0.57	1.00
S2	*A*	1	753	7	3	0.40	0.31	12	2	0.71	0.35	442	277	0.23	0.65	1	0	0	0.00	0.00	23	8	0.48	0.49
		2	38	1	0	1.00	0.03	5	2	0.43	0.18	34	67	-0.33	0.09	0	0	0	0.00	0.00	4	1	0.60	0.08
	B	1	109	14	2	0.75	0.50	8	3	0.45	0.28	155	35	0.63	0.17	4	3	0	1.00	1.00	17	5	0.55	0.35
		2	52	3	2	0.20	0.16	4	4	0.00	0.20	19	69	-0.57	0.08	5	0	0	0.00	0.00	2	3	-0.20	0.08
$\sum$			952	25	7	0.56	1.00	29	11	0.45	1.00	650	448	0.18	1.00	10	3	0	1.00	1.00	46	17	0.46	1.00
S3	*A*	1	22	6	2	0.50	0.53	5	3	0.25	0.35	30	24	0.11	0.43	17	0	0	0.00	0.00	56	24	0.40	0.43
			5	0	3	-1.00	0.20	1	3	-0.50	0.17	1	6	-0.71	0.06	0	0	0	0.00	0.00	5	24	-0.66	0.15
	B		22	2	1	0.33	0.20	8	2	0.60	0.43	37	27	0.16	0.50	5	1	0	1.00	1.00	52	12	0.63	0.34
			3	1	0	1.00	0.07	1	0	1.00	0.04	1	1	0.00	0.02	0	0	0	0.00	0.00	5	10	-0.33	0.08
$\sum$			52	9	6	0.20	1.00	15	8	0.30	1.00	69	58	0.09	1.00	23	1	0	1.00	1.00	118	70	0.26	1.00
S4	*A*	1	48	3	0	1.00	0.13	13	3	0.63	0.36	44	13	0.54	0.23	5	1	0	1.00	1.00	75	9	0.79	0.36
		2	24	1	3	-0.50	0.17	6	13	-0.37	0.42	26	42	-0.24	0.27	16	0	0	0.00	0.00	28	60	-0.36	0.38
	B	1	73	9	2	0.64	0.48	5	0	1.00	0.11	46	15	0.51	0.24	5	0	0	0.00	0.00	36	6	0.71	0.18
		2	24	3	2	0.20	0.22	3	2	0.20	0.11	29	34	-0.08	0.25	0	0	0	0.00	0.00	16	3	0.68	0.08
$\sum$			169	16	7	0.20	1.00	15	8	0.30	1.00	69	58	0.09	1.00	26	1	0	1.00	1.00	155	78	0.33	1.00
S5	*A*	1	823	-	-	-	-	-	-	-	-	323	273	0.45	0.45	-	-	-	-	-	-	-	-	-
		2	96	-	-	-	-	-	-	-	-	53	38	0.16	0.07	-	-	-	-	-	-	-	-	-
	B	1	332	-	-	-	-	-	-	-	-	311	188	0.37	0.37	-	-	-	-	-	-	-	-	-
		2	71	-	-	-	-	-	-	-	-	88	58	0.21	0.11	-	-	-	-	-	-	-	-	-
$\sum$			1,332	-	-	-	-	-	-	-	-	775	557	0.16	1.00	-	-	-	-	-	-	-	-	-

Figure 5.12 Comparison of the cumulated values for the structural stability.



5.5.3.3 Cumulated data

Figure 5.12 confirms the trend of the overdue re-structuring event for S1. We found 208 events (cumulated added and removed) *depends-on* dependencies which dropped to 200 at the end of the observation period. Since several dependencies may be added or removed during one transaction, the number of events does not accurately reflect the actual number of shared symbols. Particularly, a symbol may be introduced several times using different change transactions and later removed using a single transaction. In the "added" case the symbol is counted several times, in the "removed" case the symbols is then counted only once. Consequently, it primarily reflects the number of independent change transactions required to modify the system rather than the actual number of symbols.

For set S2 the absolute numbers are much smaller with 32 and 29 respectively but also reflect the absence of an effective re-structuring event. In file-set S3 we have a maximum of 73 which dropped after an effective re-structuring event down to 48. The reason for its asymmetry with respect to the 50% mark is that one subset can introduce more symbols than the other subset. From the graph in Figure 5.12 can be seen, that all selected file-sets reached after an approximately linear growth from the project begin their maximum in time-slot 28 which is about at release 0.9.7.

5.5.3.4 Qualitative evaluation

Following we provide our interpretation about the structural dependencies extracted from the source code changes with respect to the three selected file-sets.

Set S1: the visualization for S1 confirms roughly the growing number of structural dependencies over the whole project duration without any major re-structuring events or phases of stabilization. On a coarse level, three phases can be distinguished: (1) before time-slot 21 the development showed a continuously increasing trend in structural dependencies; (2) from 21 to 26 a number of new interfaces were added to the system causing also a *day-fly* (Bug 104336); (3) lesser activities of added and removed interfaces. Since this part of the system mainly indicates a growing trend in all structural aspects, it confirms our *God-class* hypothesis about *nsCSSFrameConstructor.cpp*. This can be also underpinned via the following quote from the *Mozilla* web page[1], which indicates that the authors were aware about architectural

[1] *www.mozilla.org/roadmap/roadmap-02-Apr-2003.html* [June 2006]

shortcomings: *"Gecko*[2] *needs to support emerging standards [...] without everyone having to hack into* nsCSSFrameConstructor.cpp*. It should be possible [...] to avoid an explosion of new C++ code and rendering object bloat."* A solution has not been proposed until version 1.7 of the *Mozilla Application Suite.* In version 1.8 they have "corrected" the problem by moving the file in the repository from *layout/html/style/src* to *layout/base*. The file itself has grown between these to version by another 500 source lines and has now a total of 13,885 source lines. Another example for the instability of architecture is the introduction of *nsIRuleNode* (Bug 78695) in time-slot 22 and its removal in time-slot 26. These events are classified as *day-fly* anti-pattern having an affect on *nsCSSFrameConstructor.cpp* and the other files of the selected file-set S1. According to the problem report (Bug 104336) the fix was due to some performance issues introduced with the component *nsIRuleNode* in time-slot 22. The applied fix in time-slot 26 caused the introduction of the file *nsRuleNode* which was used instead to avoid the overhead of the component model.

Set S2: in this file-set we compare *nsHTMLContentSink.cpp* against the other files of this set. Interesting in this set is the high number of *uses* facts which are related to HTML code parsing causing the high number of changes basically in the files from *htmlparser*. These structural dependencies were removed in time-slots 41 and 45, respectively, first from *htmlparser* ($S2_A$) and then from *nsHTMLContentSink.cpp* ($S2_B$).

Set S3: for the third third file-set we used one file-pair *nsXULDocument.cpp* ($S3_A$) from the sub-module *content/xul/document* and *nsPresShell.cpp* ($S3_B$) from the sub-module *layout/html/base*. The most frequent co-change types we detected were code and signature changes followed by build architecture, component interface and user interface changes. The resulting diagram is depicted in Figure 5.10 (some dependencies are omitted due to space limitations) with the following interesting change events:

- (A) indicates the provision of *nsPresShell.cpp* and its usage by the second file *nsXULDocument.cpp* which has to be interpreted as a direct structural dependency;
- (B) shows a major re-structuring event since many labels (upper diagram) and words (lower diagram) were removed at the same time;
- (C) is indicative in the word-view and shows some removals and updates;
- (D) marks another global *re-structuring* event which had also had a system-wide effect on many other files;
- (E) and (F) mark two interesting events with respect to the *daisy-chain* pattern. Both dependencies were first used by *nsXULDocument.cpp* and then by *nsIPresShell.cpp*;
- (G) indicates a *pulsar* for the dependency *nsIDOMXULDocument* which may point out test-code or doubt about the overall architecture.
- (H) *skipjack* after re-structuring (*nsContentUtils*, *nsIParser*).

Set S4: this pair of file-sets comprises the features fHttp and fHttps and relates to the independence of the security layer from the application protocol. It is discussed in Section 5.5.3.5.

Set S5: relates to cross language analysis with *EvoGraph* of the graphical user interface of the *Mozilla Application Suite* and is discussed in detail in Section 5.5.3.6.

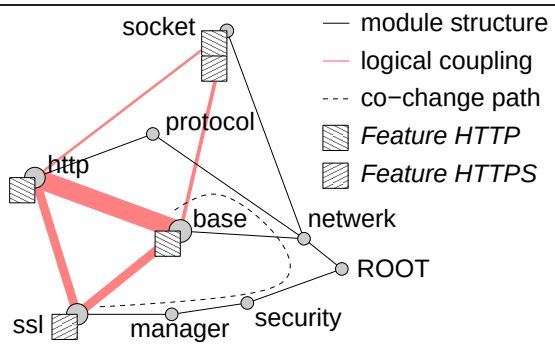
5.5.3.5 Analyzing features with EvoGraph

For the discussion of detailed structural analysis of feature evolution, we use again the two features fHttp and fHttps. A schema with their logical coupling is depicted in Figure 5.13 and the detected structural dependencies from source code change analysis in Figure 5.14.

One of the more interesting file pairs is *nsSocketTransport.cpp* from *netwerk/base* and *nsNSSIOLayer.cpp* from *security/manager/ssl*. In the first co-change transaction the SSL part for the fHttps feature was integrated into the system. As a result of this transaction we found the symbol *nsISSLSocketControl* in the source code changes of *nsSocketTransport.cpp*. For the second file–*nsNSSIOLayer.cpp*–we were able to find *nsISSLSocketControl* in the vector of identifiers derived from the change transaction and thus found a new depends-on relationship in its corresponding include file. Horizontal mining of *nsSocketTransport.cpp* reveals that the interface *nsISSLSocketControl* has been added in this change transaction. For the second

[2] Gecko is *Mozilla*'s rendering engine

Figure 5.13 Stickiness view of the features fHttp and fHttps with respect to the module structure. ROOT designates the root node of the module tree.



file we found that the dependency on this interface was added during this transaction and has not been removed again. In contrast to the first file, this file is actively maintained in current releases.

Summarizing the results for this file pair, we found six transactions which also comprised changes of component interfaces. Two of the transactions can been seen as costly since the build architecture had to be changed as well. One change transaction involved also a modification of the user interface, which can require extra effort for manual testing of the changes. Another quantitative interesting result is that, about 50% of the changes concerning the common strings were introduced in separate change transactions.

To summarize, fHttps had been added in a later stage and maintained a number of structural dependencies for about two years. With the global re-engineering event (D) the structural dependencies have been removed almost completely (see Figure 5.14).

5.5.3.6 Cross language level analysis

Another aspect of the *EvoGraph* approach is its capability to offer support for the analysis of cross language dependencies as well. Due to the lack of a file-pair with high stickiness, we analyze the file-pair *navigator.xul* and *navigator.js* from the module *xpfe*.

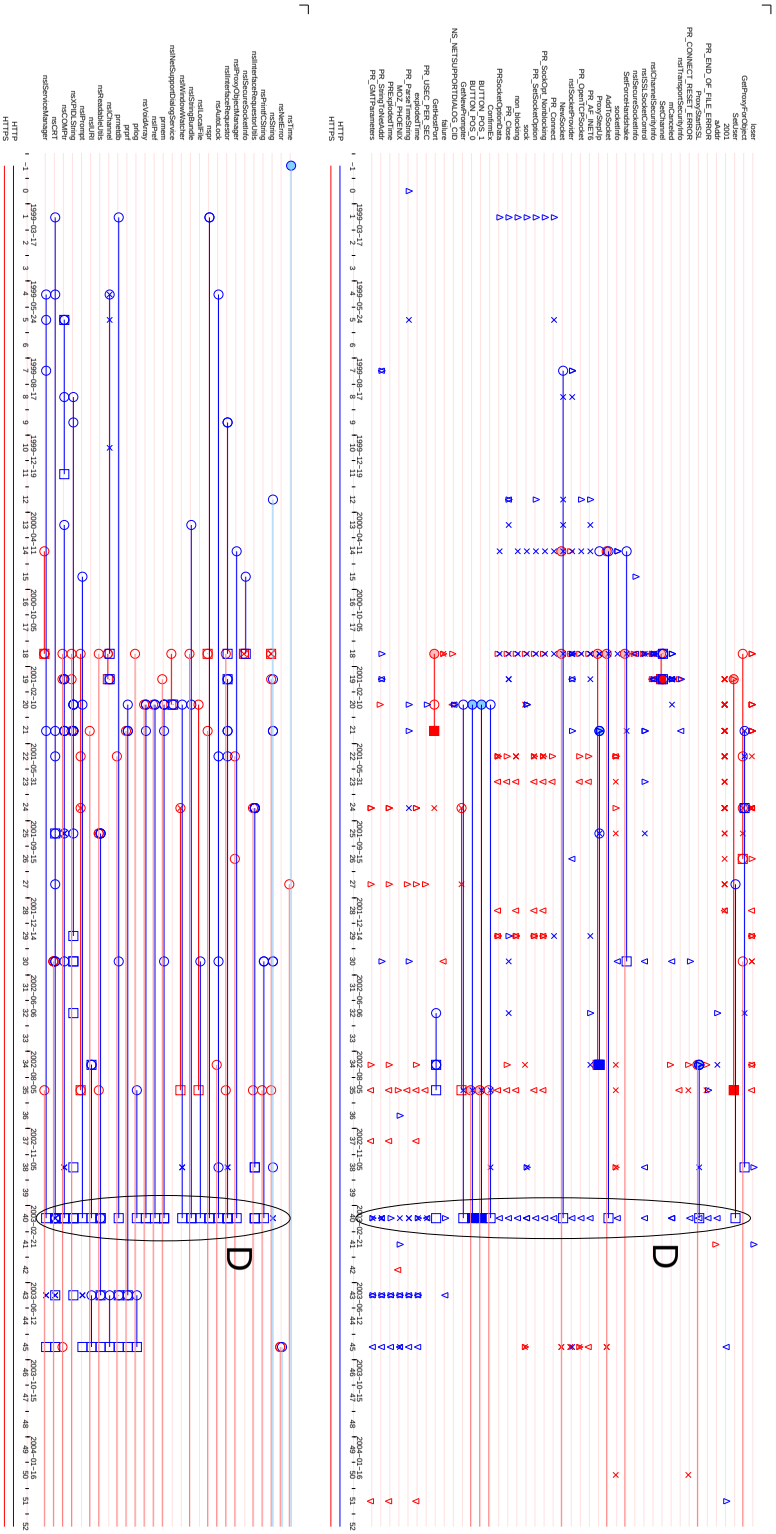
The files are part of *Mozilla*'s GUI which consists of a number of user-interface declarations written in XML and underlying clue code written in JavaScript. These directives are interpreted via an internal shell of the browser. A new full functional browser instance within an existing browser window can be created by pasting the following URL *chrome://navigator/content/navigator.xul* into the browser window. Since the were developed together with the whole *Mozilla Application Suite* they have also a very low stickiness, but we detected virtually no dependencies in the label-view. In the words-view they share a large number of identifiers which are responsible for the high adhesion.

As depicted in Figure 5.15 the curve for the cumulated values indicates an increasing number of shared symbols without re-structuring events. This type of curve is typical for file-pairs residing in the same directory. Though the number of added symbols does not seem to have a limiting factor, the curve is flattening towards to the latest releases. Conspicuous in the diagram is that a few symbols are more frequently changed by one file and other symbols typically by the other file. For instance, "oncommand" is about 100 times more frequently used by *navigator.xul* in the different source code changes than by *navigator.js*. As counter example "getElementById" appears about 10 times more frequently in *navigator.js* than by *navigator.xul*.

A frequent occurring pattern is the day-fly anti-pattern, indicating possible incorrect approaches or other mistakes. For instance, a naming problem occurred with function "addToUrlbarHistory" which also existed as "AddToUrlbarHistory". Another day-fly example is "popupBlockerMenuCommand" (Bug 198846) which has been removed in the subsequent revision due to a security problem (Bug 235457).

Also with other files from the system a number of logical couplings exist. This indicates the simultaneous development of the *Mozilla* base system and its application instance as the files *navigator.js* and *navigator.xul*. Table 5.11 shows an excerpt of logically-coupled files with still detectable couplings be-

Figure 5.14 Commonalities between features fHttp and fHttps.



(a) label-view (b) word-view

Figure 5.15 Structural changes (solid line) and cumulated values (dashed line) for file-set S5.

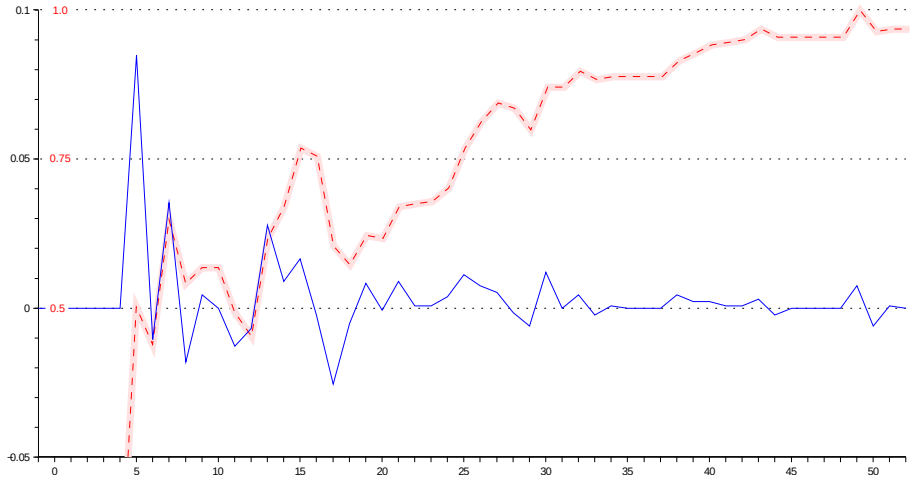


Table 5.11 Logical coupling between user interface files and the underlying base system.

LC	Module	File	Module	File	#
A	*xpfe*	*nsBrowserInstance.cpp*	*xpfe*	*navigator.js*	51
S	*docshell*	*nsDocShell.cpp*	*xpfe*	*navigator.js*	33
S	*dom*	*nsGlobalWindow.cpp*	*xpfe*	*navigator.js*	24
A	*xpfe*	*utilityOverlay.js*	*xpfe*	*navigator.js*	21
A	*xpfe*	*nsBrowserInstance.cpp*	*xpfe*	*navigator.xul*	18
A	*xpfe*	*nsContextMenu.js*	*xpfe*	*navigator.js*	16
S	*docshell*	*nsWebShell.cpp*	*xpfe*	*navigator.js*	16
A	*xpfe*	*nsWebShellWindow.cpp*	*xpfe*	*navigator.js*	16
S	*content*	*nsDocumentViewer.cpp*	*xpfe*	*navigator.js*	12
S	*dom*	*nsGlobalWindow.cpp*	*xpfe*	*navigator.xul*	11
A	*xpfe*	*nsWebShellWindow.cpp*	*xpfe*	*navigator.xul*	11
A	*xpfe*	*utilityOverlay.js*	*xpfe*	*navigator.xul*	11
…	…	…	…	…	…

tween the entities. As expected, the file with the strongest logical coupling *nsBrowserInstance.cpp* stems from the same module.

From the *labe-* and *word*-view depicted in Figure 5.16 we can deduce that a number of direct structural dependencies exist which are the cause for the frequent pairwise changes. In contrast to some other previously inspected file-sets, a major re-structuring event is not observable in this configuration.

5.5.3.7 Discussion

All file-sets maintain a high indirect structural dependency which stems from the large number of common component interfaces they use. Moreover, S3 has also a direct structural dependency which makes things worse. As a consequence, both files can be considered to have an instable structural dependency and will exhibit co-change behavior in the future as well. Though the developers are aware about some structural shortcomings, a solution has not been realized so far. We have identified a number of interesting evolutionary properties of *Mozilla* with the *EvoGraph* approach during the evaluation of the case study:

- the graphs provides feedback about the structural dependencies which not only allows to identify the responsible source code entity but also allows to identify interesting change patterns;
- the visualizations provide a comprehensive overview about past development activities;
- re-engineering events can be identified and can assessed to identify successful or unsuccessful measurements;
- tracing source code links into GUI related files provides insight into structural dependencies between different implementation concepts which are not captured by other software analysis programs; and
- from the mining perspective, change transactions exhibit a more complex and heterogenous structure with respect to dependency exposure than expected. This increases the effort to fully track and understand the evolutionary impact of source code changes.

In the next section we discus the actual effect on the executable code of source code changes applied on the *Mozilla Application Suite* with respect to selected versions.

5.6 EvoTrace - observing evolution via runtime data

As in the previous chapters we continue to use the *Mozilla Application Suite* as a representative and challenging case study. Major reasons for that are the already existing *Release History Database* with structural and evolutionary information. Furthermore, results of this work have been integrated into the *Release History Database* to further augment the exploration of the software evolution information space.

The used snapshots for this evaluation is based on version 1.7 (released 2004-06-18) and version 1.4 (released 2003-07-01) of the *Mozilla* code base. Version 1.7 is used because it is the latest version we have release history data. Releases prior to 1.4 require an outdated version of the GNU compiler collection (GCC), thus earlier releases are not compilable with our currently installed version of GCC (which was version 3.3.3 at that time).

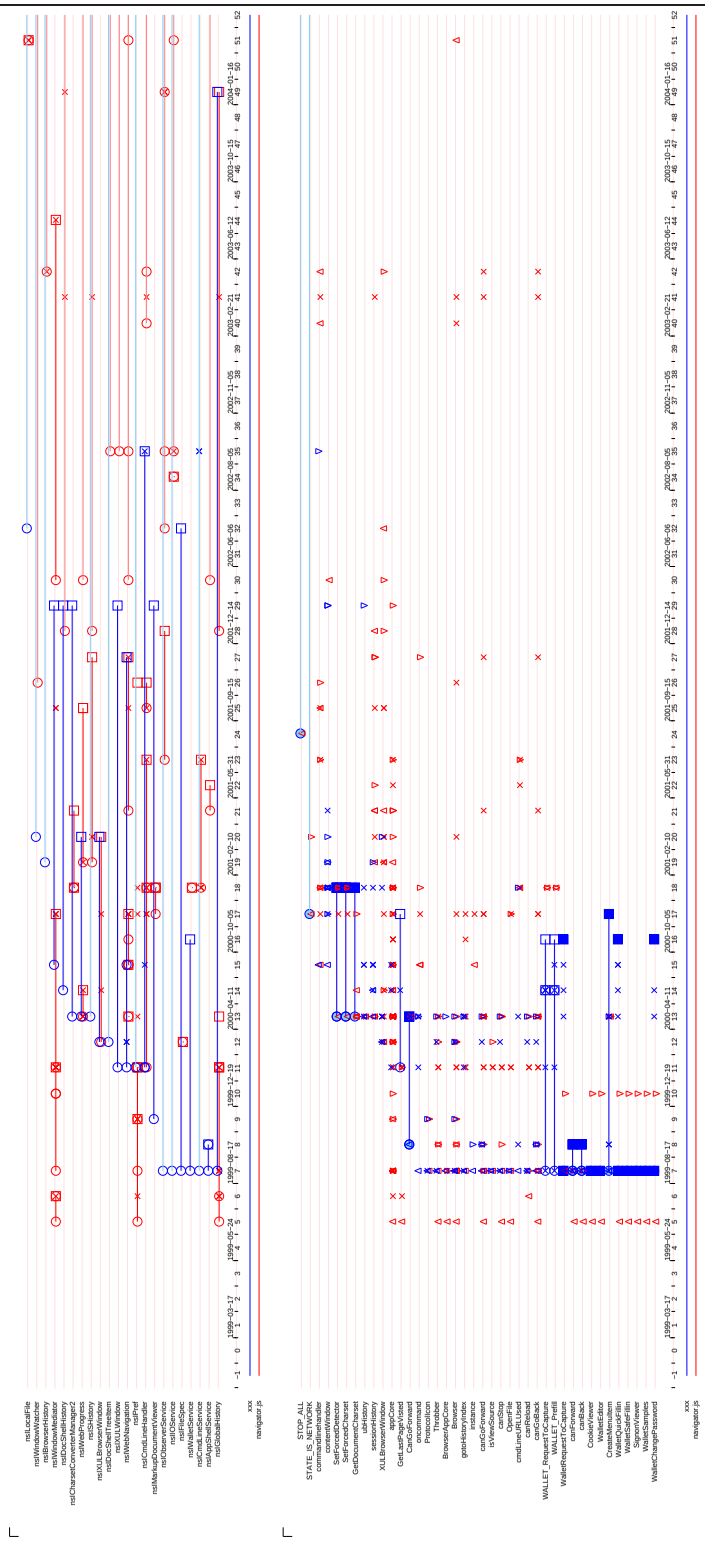
To demonstrate the *EvoTrace* approach we use a subset of the available *Mozilla* modules. They are related to web content representation and layout and appeared already in the previous sections. Table 5.12 lists the selected modules.

5.6.1 Data collection

Before data collection can start some preparation work of the testee has to be done. Both source code versions of *Mozilla* are instrumented via the `-finstrument-functions` compiler option provided by GNU compiler collection. This option generates instrumentation code for entry-to and exit-of functions. Just after function entry and just before function exit, the following profiling function will be called with the address of the current function and its call site [119]. Returns from methods can be recorded with a similar C function.

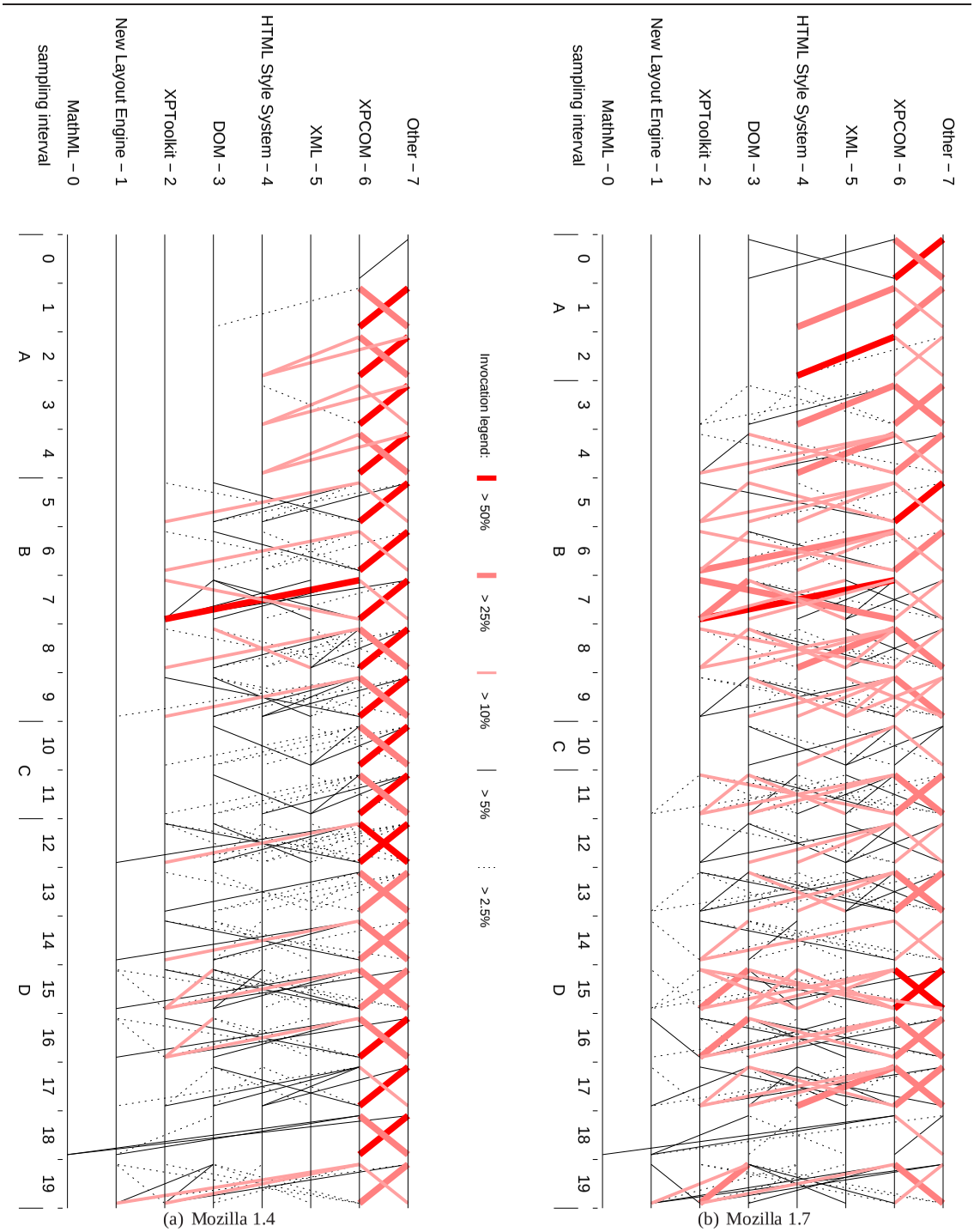
To avoid conflicts with instrumentation functions the attribute `__no_instrument_function__` has to be applied. This prevents their recursive invocation. Another source of conflict is *Mozilla*'s thread library

Figure 5.16 Commonalities between nsBrowserInstance.cpp and navigator.js.



(a) label-view (b) word-view

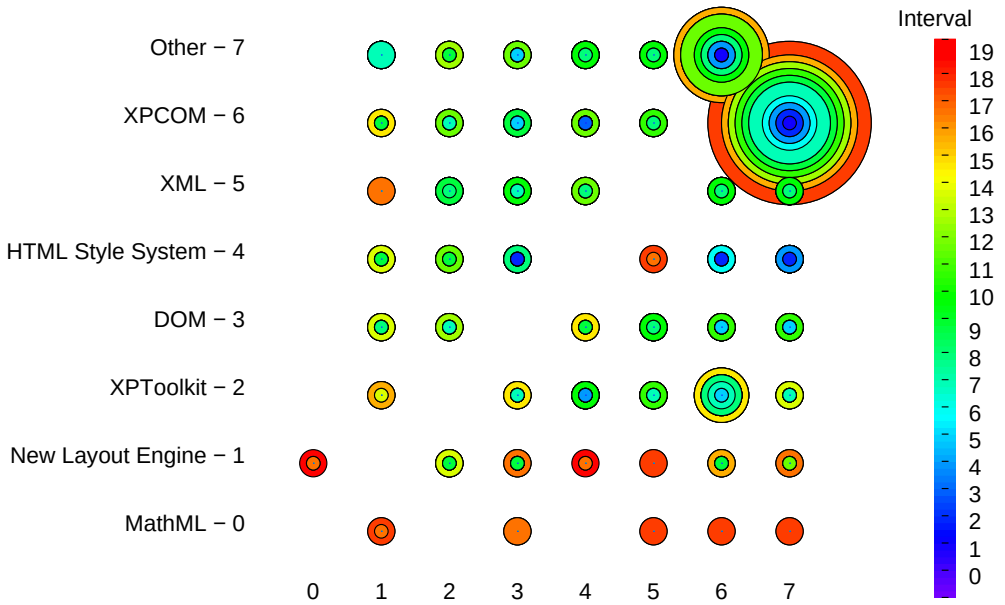
Figure 5.17 Execution trace for Mozilla modules as Gantt diagram.



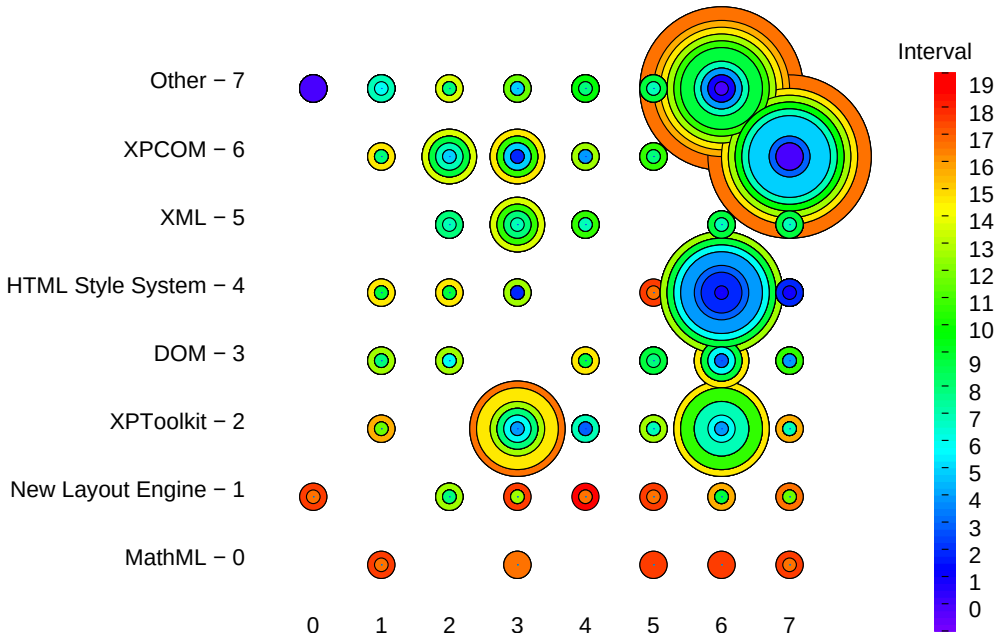
(a) Mozilla 1.4

(b) Mozilla 1.7

Figure 5.18 Interval-based tree-ring invocations for Mozilla modules as matrix view.



(a) Mozilla 1.4



(b) Mozilla 1.7

Figure 5.19 DOM related invocations in Mozilla 1.4 (left) and 1.7 (right).

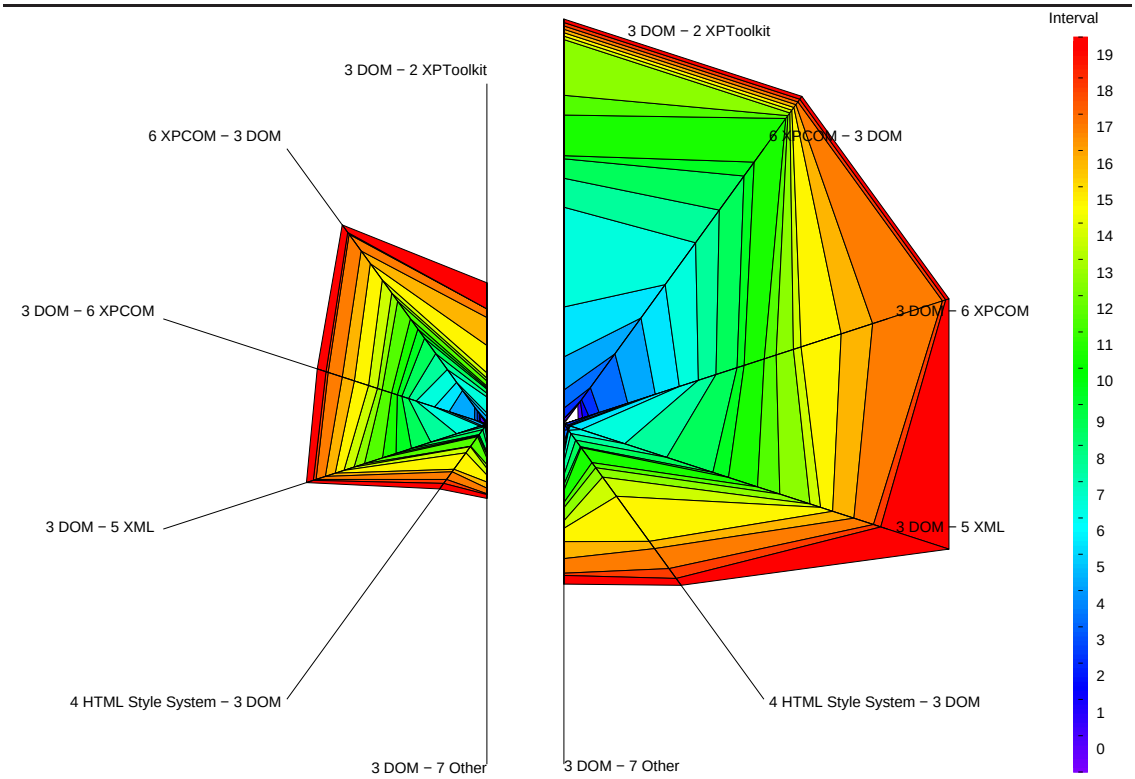


Table 5.12 Selected Mozilla modules and their source code directories.

Module ID	Module	Source Directories
1037	MathML	*layout/mathml*
1046	New Layout Engine	*layout/base*, *layout/build*, *layout/html*
1086	XPToolkit	*content/xul*, *layout/xul*
1010	Document Object Model	*dom*, *content/base*, *content/events*, *content/html/content*, *content/html/document*
1045	New HTML Style System	*content/html/style*, *content/shared*
1077	XML	*content/xml*, *expat*, *extensions/xmlextras*
1087	XSLT	*content/xsl*, *extensions/transformiix*
1081	XPCOM	*xpcom*

Listing 5.4 Telling the C compiler not to instrument the designated function.

```
void __attribute__ (( __no_instrument_function__ ))
    __cyg_profile_func_enter (void * callee , void * caller )
```

Table 5.13 Basic results obtained from trace data.

Mozilla	Version 1.4	Version 1.7	Δ (% between versions)
Binary size	82,109,017	101,012,842	+23
Number of events	23,878,728	18,822,452	-21
Callee addresses	12,077	11,644	-3.6
Caller addresses	41,962	37,011	-12
Number of threads	4	5	+25
Deepest call nesting	153	164	+7.2
Number of methods	11,940	11,563	-3.2
Number of files	868	850	-2.1
Files from modules	403	396	-1.7

`nsprpub`. We require some functions of this library to determine the thread context under which the instrumentation function is executed. For this library we completely disabled code instrumentation. After this preparation step, both program versions were compiled using the same compiler and configuration options.

To avoid interference through user interactions, we implemented a shell script which automatically starts the application with the specified test-parameters and terminates the application after a predefined timeout period. As test-scenario we use a copy of a page of the W3C's *MathML test suite* which we placed on our web server[3]. Differences in the resulting execution traces due to network indeterministic can be neglected since the selected modules are not related to network communication. As timeout when the application shall receive the `QUIT` signal we determined one second where no events are recorded to be sufficient.

Additionally, pre-loading of web-pages, changes to the page cache and the URL visit history are causes for differences in the execution traces, especially when running different versions of the program. To minimize these impacts, we used *three test-runs* in a row whereas only the results of the last one is used (the test-runs two and three produced traces with a similar number of events). To avoid conflicting interactions with the window manager of our test-system, we used a separate X-Window server without any window manager functionality. During test-runs the application window is redirected to this separate server while trace data are stored on the local disk drive.

Currently, we used the C `printf`-function to record each *enter* or *exit* event. The snippet below depicts the data format produced by the instrumentation function:

```
e0x8cede20m0x8cedf09t0x8f26548
```

Four types of information are recorded: (1) the event-type (*enter* or *exit*); (2) the callee address; (3) the caller address (starting at 'm'); and (4) the thread context (starting at 't').

5.6.2 Post-processing and quantitative results

After the import of the raw data into the database via a Perl script, we can obtain first quantitative results with simple SQL queries. The results are listed in Table 5.13 for both *Mozilla* versions.

The binary file of version 1.7 is large compared to version 1.4 but the code seems to be leaner and produces less execution trace events. Even though listed as exact numbers, the *number of events* vary slightly between test-runs since network communication or the OS timing is not deterministic.

[3] *http://www.infosys.tuwien.ac.at/staff/mf/test/iwpc05/math3.xml*

The number of different start addresses of invocations found in the execution traces is given by *callee addresses*. This differs from *number of methods*—number of different methods signatures found in object files—which is based on the symbol information delivered by `nm`. Differences originate from C++ language constructs and internal management tasks of the runtime library. The *caller addresses* lists the number of different addresses from where methods have been invoked. To assign traces to the correct thread context we record the thread ID at each event. Consequently, the *number of threads* reflects the total number of different IDs found.

One aspect not covered by "traditional" profiling is the nesting-level. With *deepest call nesting* we give the deepest level of invocations found in the execution traces. During the import phase the *callee* address information is combined with the symbol information from object files. Here, the *number of files* represents the number of successful maps to source files. In a post-processing phase, we then identified those files which belong to the modules we are interested in (*files from modules*). This speeds up later data analysis.

5.6.3 Visualization

After the generation, filtering and first quantitative evaluation of the test-data, we visualize the results for evolution tracking. As described in the previous section, we divide the execution traces into twenty different intervals for sub-sampling. This is sufficiently small to distinguish different types of interaction phases but is large enough to create "readable" visualizations. While the interval size is more relevant for the first diagram type we present here, it is of minor relevance for the other two.

5.6.3.1 Gantt diagram

One well-known form for visualization of execution traces are Gantt diagrams which are suited to study interactions on a very fine-grained level. Since our *EvoTrace* approach is designed to reveal coarse changes in system interaction, we use a "reduced" form of the Gantt diagram type where the invocations are sketched. In Figure 5.17 this modified diagram type is depicted with the filtered invocation sequences of *Mozilla* 1.4 (a) and 1.7 (b) respectively. In both diagrams the invocation frequencies between modules are divided into six classes: $> 50\%$, $> 25\%$, $> 10\%$, $> 5\%$, $> 2.5\%$ and $\leqq 2.5\%$ whereas invocations of the last class are not shown. Invocations are depicted as lines with different shapes representing their frequency between modules.

When visually comparing diagrams (a) and (b) the differences in invocation intensity between the modules Other and XPCOM are significant. This was surprising, since we did not expect such extreme changes. Interesting to see are also the mutual invocations between Other and XPCOM. But this is an expected result since Other contains all other modules we did not explicitly identify.

Roughly, four phases can be distinguished: (A) prelude; (B) user interface related activities (XPToolkit is the cross-platform user interface); (C) an intermediary phase; and (D) content related activities including MathML. The main differences are that phase (A) begins in version 1.7 two time-slots earlier compared to version 1.4 and that the intermediary phase (C) can be clearly identified. Remarkable is also the strong communication path in slot 7 from module XPCOM to XPToolkit which appears in both versions.

5.6.3.2 Matrix view

To overcome the problem of clutter in the Gantt diagram, we use a specific matrix view, which supports the visualization of invocations as cross product between modules. Callers are placed on the horizontal axis and callees are placed on the vertical axis. For instance, to find the invocations from XPCOM to HTML Style System can be found by going to column 6 and move up till row 4. During the development of this view we noticed, that presentation quality suffers from the wide spread of invocation frequencies that can differ by an order of magnitude of 5. As solution, we introduced five frequency classes according to the overall maximum number of invocations. Each class has a fixed size so we get data sets with maximum value $\in [0.2, 0.4, 0.6, 0.8, 1.0]$. The data are then scaled to the desired size during diagram generation. As the forth dimension in our visualizations we have the time dimension. We decided to use a *tree-ring scheme* and rainbow colors to depict the twenty intervals: dark blue indicates the first interval—most inner ring—and red indicates the last interval. Since the values are scaled to different maxima—one maximum for each

Mozilla version—sizes between both diagrams must be compared via absolute values from the database. A brief comparison indicates that the communication in version 1.7 is more distributed compared to version 1.4 of *Mozilla*. In contrast to Figure 5.17 where the changes in the invocation frequencies are not directly recognizable, the matrix type view depicted in Figure 5.18 supports perception of these changes in an intuitive way.

Striking is the high number of invocations between XPCOM and Other in Figure 5.18.(a) whereas Figure 5.18.(b) shows a more "balanced" characteristic. Another interesting result is that communication starts earlier in version 1.7 (e.g., XPCOM - HTML Style System) compared to the predecessor version (which is also supported by the Gantt diagrams). This can be interpreted in such a way, that the system has been optimized and web pages are now delivered faster to the user.

Next, we give a more detailed view of one selected software module with respect to invocations with other modules.

5.6.3.3 Detailed module view using Kiviat diagrams

As result of the *EvoTrace* approach, we obtain multidimensional data sets. To overcome some of the limitations of the previous views, we decided to use Kiviat diagrams for a detailed view on the communication between modules. Two diagrams covering a range of 180° each, face by face, allow a quick comparison of specific module data between two releases. Based on the experiences with the wide value range we sorted values in ascending order and limited the result set to the six most frequent invoked module pairs. Further modifications concern the scaling of the data sets during diagram generation. For data representation we use a 4-dimensional dataset. The actual value of each data point in the diagram is determined by the following scaling formula:

$$v_{k,a,b,t_s} = \begin{cases} \frac{2}{\max} F & \text{if } 2F < 1 \\ 1 & \text{if } 2F = 1 \\ 1 + 0.01 * s & \text{if } 2F > 1 \end{cases} \tag{5.4}$$

whereas $\max$ is the overall maximum of F, and

$$F = \sum_{s \leqq n} f_{k,a,b,s} \tag{5.5}$$

is the cumulated value if invocations between module M_a and M_b for version k of *Mozilla* over the timeslots $t_s \leqq n$ ($n \in \{0, 1, ..., 19\}$). Division of the maximum by a constant factor together with the $2F > 1$ branch, reduces the biasing effect of "spikes" in the final diagram.

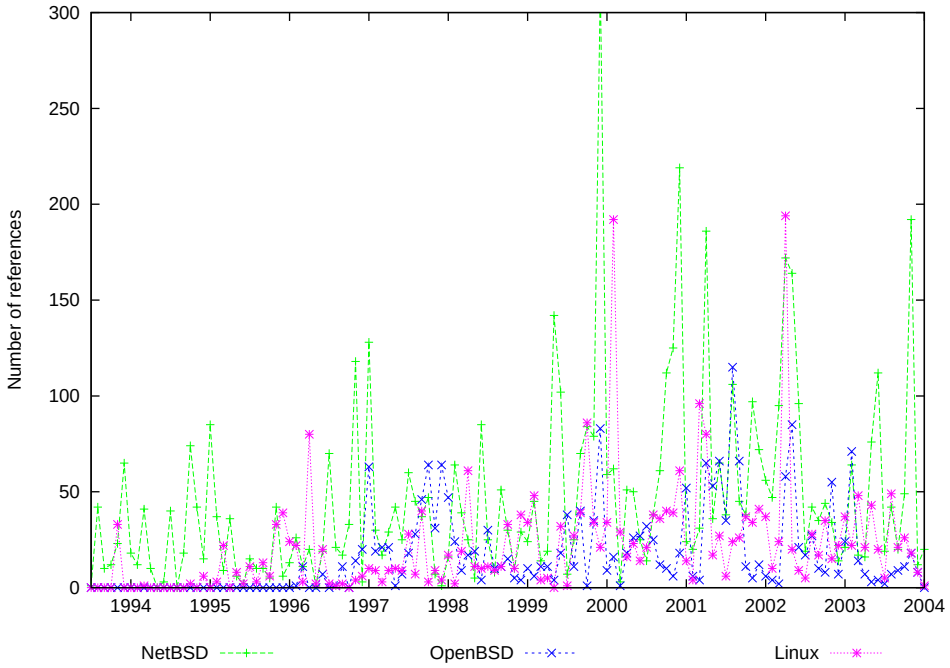
The resulting diagram for module DOM is depicted in Figure 5.19 whereas the DOM - XPToolkit, XPCOM - DOM, and DOM - XPCOM are reduced in size ($2F > 1$ branch). In contrast to the matrix view, the data sets for the different releases are scaled with a common factor. Thus both sides of of the diagram are directly comparable. Our example graph indicates for the modules DOM and XPToolkit that communication has doubled. Especially during the center period (light area) the increment was substantial. This perception is supported by data from the database where 50,190 invocations for version 1.4 and 105,001 for version 1.7 have been recorded. Further interesting are the delays when communication starts with relevant modules. Examples for early communication (inner dark area) in version 1.4 are DOM - XPCOM (both directions) and DOM - Other, respectively. Compared to version 1.7 no significant changes can be found for these modules. As a counter example with respect to scheduling information, we refer to the pair HTML Style System - DOM where communication starts late in both program versions.

Another interesting area of application is the deduction of *uses* relationships. This is facilitated by this diagram type, since results are sorted by frequency and further subdivided than in the Gantt diagram or Matrix view. As depicted in Figure 5.19 most communication takes place in a single direction between modules. Counter example is DOM - XPCOM where communication is shown to happen in both directions.

5.6.4 Discussion

With a traditional database approach large amounts of trace data can be handled efficiently, the database queries are simple to implement and access via standard SQL query interface for third party tools is pos-

Figure 5.20 Number of references to keywords *NetBSD*, *OpenBSD*, and *Linux* found in *FreeBSD* change logs.



sible. Another advantage is that storing the program traces in the *Release History Database* supports fast retrieval and detection of a system's interaction patterns without losing context related detail information. During our experiments access speed was not in issue. The detection of invocation sequences of a single trace with more than $19 \cdot 10^6$ events using a Java program and MySQL database on a Pentium 4, 2.8GHz, 1GB takes less than 5 minutes, which we considered reasonably fast. If a speed up for pattern detection is required, the problem space can be nicely partitioned via invocation-levels.

Though some of the results can be achieved with data from conventional profiling as well, focus of the *EvoTrace* approach is the evaluation of program traces for evolution analysis. Our visualizations provide insights into changes on arbitrary detailed level to track the changes between system releases.

5.7 EvoFamily - identifying commonalities in product families

For the evaluation of the approach, we decided to we study the evolution and commonalities of three variants of the BSD (Berkeley Software Distribution), a large open source operating system. The research questions we tackle are concerned with how to generate high-level views of the system discovering and indicating evolutionary highlights. The release history for the three selected variants represents about 8.5GB of data and 10 years of active development.

The selected three variants—*FreeBSD*, *NetBSD*, and *OpenBSD*—of BSD are large software systems consisting of an operating system kernel and a number of external programs such as *ls*, *passwd*, the GNU Compiler Collection (GCC), or the X windows system. These variants have between 4,800 for the *OpenBSD* variant and 8,000 directories for the *NetBSD* variant. The number of files varies between 30,000 (*FreeBSD*) and about 68,000 (*NetBSD*). They are long-lived, actively maintained software systems representing about 8.5GB of data stored in three different repositories. Furthermore, release information is available as CVS data for all three variants with direct access to the current repositories. The systems itself possess different characteristics which can be described as follows: The *FreeBSD* [4] projects aims to

[4] *http://www.freebsd.org/* [30 June 2006]

Figure 5.21 Number of references to keywords *FreeBSD*, *OpenBSD*, and *Linux* found in *NetBSD* change logs.

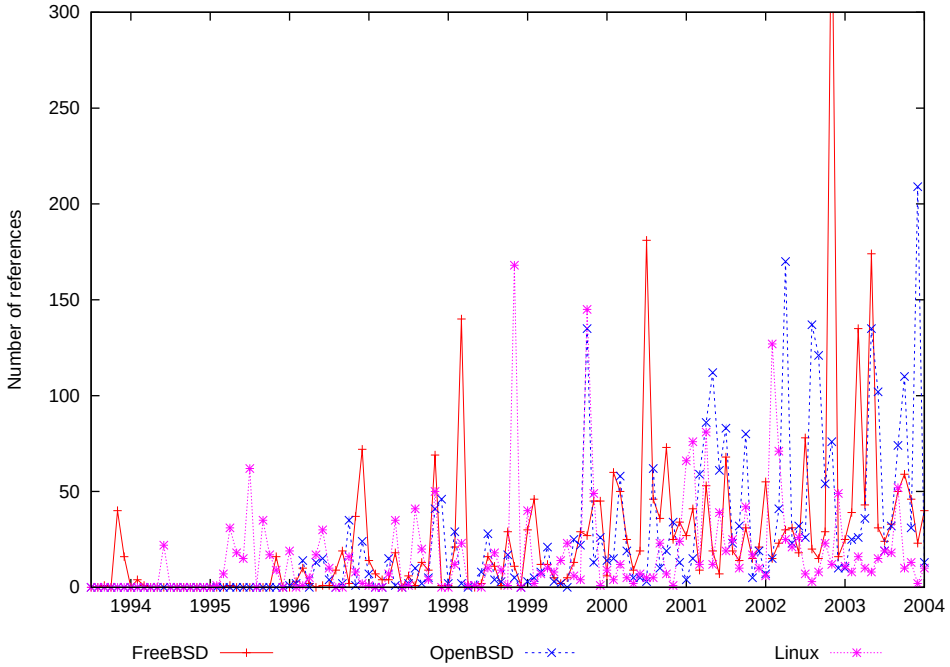


Figure 5.22 Number of references to keywords *FreeBSD*, *NetBSD*, and *Linux* found in *OpenBSD* change logs.

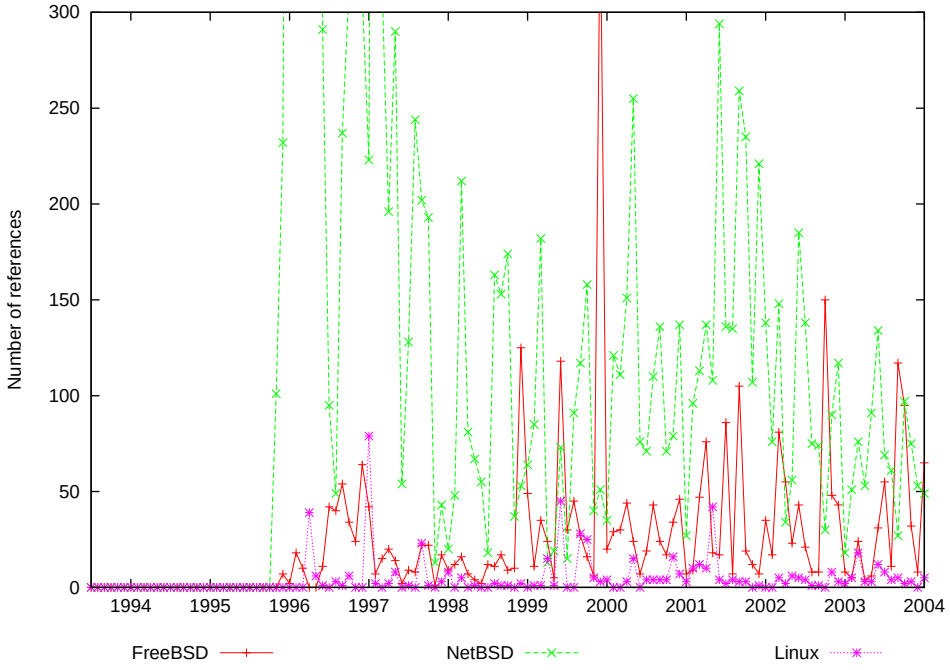
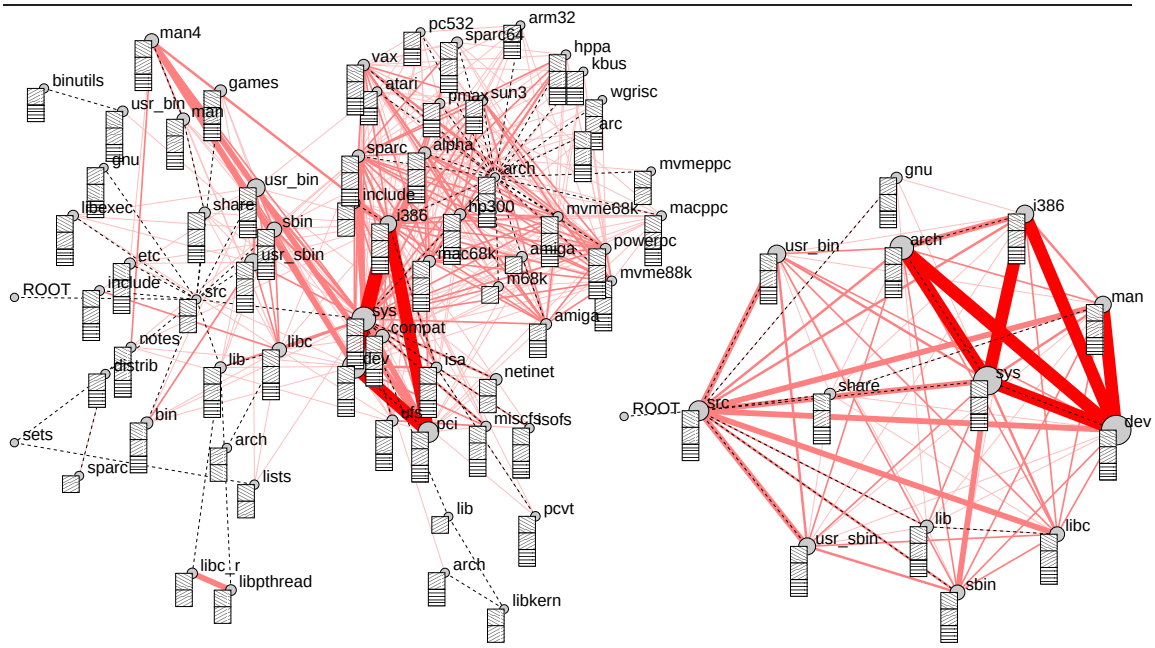


Figure 5.23 Change coupling between modules of the source code structure of the OpenBSD system with emphasize on the module structure.



(a) Change coupling complete system. (b) Change coupling with focus on source modules.

Table 5.14 Common files in different BSD variants.

Variant	Variant	all modules	src/sys/ only
FreeBSD	*NetBSD*	3,810	1,333
FreeBSD	*OpenBSD*	3,839	1,079
NetBSD	*OpenBSD*	6,969	6,847

be more user application centric and thus it can be seen as desktop OS rather than server platform. Its first release was in December 1993. *NetBSD* [5] is targeted onto portability and supports more than 10 different CPU types with together more than 50 different hardware platforms. Among them are exotic platforms such as *Acorn*, *Amiga*, *Atari* or *VAX*.Its first release was in October 1994. As representative of a server platform the aim of the *OpenBSD* [6] project lies on security and the integration of cryptography. Its first release was in October 1996. While *NetBSD* and *FreeBSD* were directly derived from the *4.3BSD* branch, *OpenBSD* was derived from the *NetBSD* branch in October 1995.

5.7.1 Quantitative comparison

First we give a quantitative comparison of the number of artifacts which are common for the different systems. To determine the number of common C files in the different *Release History Database*s we use multi-database SQL queries which can be nicely written in *MySQL* as whereas `rbsnet` and `rbsdopen` are the *Release History Database*s for *NetBSD* and *OpenBSD*, respectively.

Table 5.14 shows the result for the different variants. While column "*all modules*" indicates the total number of common files found, column "*src/sys only*" indicates the common files within this particular subtree. Interesting is the high number of artifacts which are common in *NetBSD* and *OpenBSD*. This can be explained by the fact that *OpenBSD* was derived from *NetBSD* as mentioned previously.

[5] *http://www.netbsd.org/* [30 June 2006]
[6] *http://www.openbsd.org/* [30 June 2006]

Listing 5.5 Counting common C files in the different projects.

```
SELECT COUNT(*)
  FROM rbsdnet.cvsitem n, rbsdopen.cvsitem  o
  WHERE n.rcsfile=o.rcsfile  AND n.rcsfile  REGEXP "\\.c$";
```

Table 5.15 Information flow between variants of the BSD systems based on lexical search.

Variant	Keyword	all revisions	revision > 1.1
FreeBSD	*netbsd*	5,131	3,577
	openbsd	2,729	1,353
	linux	1,791	1,387
NetBSD	*freebsd*	2,852	2,186
	openbsd	2,679	2,224
	linux	1,547	1,125
OpenBSD	*freebsd*	2,406	1,933
	netbsd	16,802	7,423
	linux	775	463

5.7.2 Change report text analysis

As substitution for a detailed text and code clone analysis, we use keywords which were frequently used by the program authors and recorded in change reports. As useful keywords we identified *freebsd*, *netbsd*, *openbsd*, and interestingly *linux*.

Table 5.15 lists the number of referenced artifacts between product variants based on a lexical search for the chosen keywords in the change logs. Column one lists the name of the product variant used to retrieve the change logs and column two the respective keyword. Column three entitled "*all revisions*" lists the number of distinct artifacts found in the *Release History Database* having change logs with the specified keyword. Column four titled "*revision* > 1.1" lists the number of distinct artifacts found in the *Release History Database* having change logs with the specified keyword and not having a revision number of "1.1" (which denotes the initial revision). The significant difference between the values in column three and four can be interpreted in such a way, that a larger number of files were imported from other systems and further maintenance is decoupled from the originating version.

An interesting property when relying on certain keywords is its implicit specification of information flow. Just the simple appearance of the word *FreeBSD* in *NetBSD*'s change log message desribes the information flow from *FreeBSD* in *NetBSD*. In contrast to this, the mining for similarties in the change log messages requires timely information as well to determine the direction of information flow.

5.7.3 Reference distribution

During the lexical search for the given keywords we recorded in total 12,540 change logs for *FreeBSD*, 9,468 for *NetBSD*, and 20,906 for *OpenBSD*. Based on these results, Figure 5.22 depicts the distribution of references with respect to the observation period. Visually the histogram for *OpenBSD* suggest a strong decreasing trend in the information flow from other platforms into the *OpenBSD* source code repository.

Table 5.16 Linear regression for referenced keywords as $y = d + kx$ for the whole observation period, for the years 1995–2001 ($y = d_{1,2} + k_{1,2}x$) and the years 2001–2004 ($y = d_{3,3} + k_{3,3}x$).

Variant	d	k	$d_{1,2}$	$k_{1,2}$	$d_{3,3}$	$k_{3,3}$
FreeBSD	22.7	0.897	-2.67	1.46	387	-2.35
NetBSD	-22.7	1.28	-15.7	1.14	-21.3	1.31
OpenBSD	407	-2.57	543	-4.90	668	-4.48

For the other two system *FreeBSD* and *NetBSD* the results are depicted in Figure 5.20 and 5.21, respectively. Though, especially the *NetBSD* exhibits an increasing trend, we found the largest number of alien artifacts within the *OpenBSD* system. To underpin the visual perception of the growing trends we use linear regression analysis to find the dependency between the number of references and time-scale intervals. To test the development of the references over the given observation period we computed the values for the whole period and two sub-intervals:

- the first interval accounts for about 2/3 (variables $k_{1,2}$ and $d_{1,2}$) of the observation period which corresponds to the years 1995–2001;
- the second interval accounts for about the last 1/3 (variables $k_{3,3}$ and $d_{3,3}$) of the observation period which represents the last 36 months of the development history (years 2001–2004).

Table 5.16 shows the results for the three variants indicating a strong increasing trend for *FreeBSD* and *NetBSD* ($k > 0$ for both variants over the whole observation period). For *FreeBSD* this trend reverses for the last 36 months ($k_{3,3} < 0$). The low number of total change logs found for *NetBSD* and the positive trend in the change dependency of *NetBSD* suggest that large amounts of source code are still derived from the other OS variants. This perception is also supported by Table 5.15 since *NetBSD* has the highest ratio between the two counted categories "*revisions* > 1.1" and "*all revisions*". In contrast, *OpenBSD* exhibits a decreasing trend in both sub-intervals and the whole observation period starting from a high level (straight line in Figure 5.22). In the next sections we will evaluate these dependencies and provide a more detailed look onto the linkage with respect to different products.

5.7.4 Change impact analysis

To show the impact of changes onto the module structure with respect to foreign source code we selected *OpenBSD* for a closer inspection since we counted here the most keywords referencing other OS (see Table 5.15). The relevant artifacts were identified through lexical search as previously described. Based on the search results and the change log data the impact of change dependencies on the module structure is evaluated. The result of this step is depicted in the Figures 5.23(a) and 5.23(b). It shows the module structure together with change dependencies derived from the change log data. While filled circles indicate the nodes of the directory tree, shaded boxes indicate different product variants. We use ▧ as glyph for *FreeBSD*, ▨ for *NetBSD*, and ▤ is used for *Linux*. The approach for generating the layout for change dependencies information is based on Multi Dimensional Scaling (MDS) [91] which we used already to visualize to impact of problem report data onto *Mozilla*'s module structure.

To avoid cluttering of the figure with the several hundred modules of the source code package, we shifted relevant information from lower-level nodes of the nested graph structure towards the root node until a predefined threshold criterion—at least 64 references through change couplings per node—is met. The node sizes indicate the number of references found for each node and its sub-trees. While dashed lines indicate the directory structure of the source package, solid gray and black lines (pink and red on color displays) indicate the logical coupling between different parts of the system.

Figure 5.23(a) shows the dependencies between modules with emphasize on the module structure (149 nodes). The distribution of the glyphs for *FreeBSD*, *NetBSD*, and *Linux* indicates a significant impact—though decreasing trend—of the other OS variants onto the development of *OpenBSD*. Only very few modules such as *libpthread*—POSIX threads were not part of the *Linux* kernel sources—or *lists* (on the bottom left in Figure 5.23(a)) are not infected by "*Linux* virus". This wide distribution of *Linux* related change dependencies is a surprising result since we did not expect such a distribution after the quantitative analysis. Interesting as well is that change dependencies occur mainly within the src/sys sub-structure which represents the kernel related source code parts.

After filtering of less relevant modules and shifting the information to higher-level modules in the hierarchy we obtain the graph depicted in Figure 5.23(b) (14 nodes). Here, the graph layout respects the strength of coupling relationships—the stronger the coupling, the closer the nodes—between the different modules. This more comprehensible and less cluttered picture of couplings highlights the dependencies of the documentation in src/share/man, the system administration programs in src/sbin, user application programs such as *ls* in src/usr_bin and src/usr_sbin from the OS kernel related files underneath src/sys.

Table 5.17 Topmost referenced files with one of the given keywords in the change logs of *OpenBSD*.

Keyword	Count	Path
freebsd	59	*src/sys/dev/pci/files.pci*
.	52	*src/sys/dev/pci/pciide.c*
.	52	*src/sys/dev/pci/pcidevs*
.	50	*src/sys/arch/i386/i386/machdep.c*
.	48	*src/sys/conf/files*
netbsd	45	src/sys/arch/i386/i386/machdep.c
.	43	src/sys/dev/pci/pciide.c
.	39	src/sys/conf/files
.	34	src/sys/arch/i386/conf/GENERIC
.	34	src/sys/dev/pci/files.pci
linux	14	*src/sys/compat/linux/linux_socket.c*
.	14	*src/sys/compat/linux/syscalls.master*
.	13	*src/sys/compat/linux/linux_misc.c*
.	7	*src/sys/compat/linux/files.linux*
.	7	*src/share/man/man8/compat_linux.8*

Interesting to see is also the strong coupling via “foreign” source code changes between src/sys/arch/i386 and src/sys/dev since this coupling spans across the module hierarchy.

Since the size of the nodes indicates the number of relevant change entries found, we can conclude that the strongest impact of change coupling was on src/sys, src/sys/dev, src/sys/arch, and src/sys/arch/i386. Table 5.17 lists an excerpt of the topmost referenced artifacts which suggests a high information exchange with other software systems.

An example for the propagation of commonly required feature is the introduction of the PCI bus. Since this device type was not widely available at the time of the *OpenBSD* fork in 1996, support had to be added later requiring several separate changes as Table 5.17 suggests. Another interesting aspect is the relationship with *Linux*. The listing of if_wireg.h suggests that specific information about WLAN adapters are obtained from *Linux* as well.

5.7.5 Detailed change analysis

Since the three BSD variants originate from the same UNIX branch, it is to expect that also a number of source code changes exhibit the same or at least similar structure. For a manual verification we randomly selected one file which is available in all three variants. For this file—ufs_quota.c from the src/sys/ufs/ufs/ directory—we manually inspected the revision history for significant changes.

One significant change was the modification of a function call in the *FreeBSD* version of ufs_quota.c on 1994-10-06 (revision 1.2 → 1.3) resulting in eight modified source lines. The *diff*-snippet—depicted below—for the affected source code revision shows a single change of a source line. The first line indicates the removed code, whereas the third one shows the replacement code. The three dashes in-between indicate a delimiter line. In the change log we found the following comment, which indicates the reason for the

Listing 5.6 Source code change in FreeBSD.

```
< sleep((caddr_t)dq, PINOD+2);
---
> (void) tsleep((caddr_t)dq, PINOD+2, "dqsync", 0);
```

source code modification: “*Use tsleep() rather than sleep so that ’ps’ is more informative about the wait.*”

The same modification in the *NetBSD* version has been applied on 2000-05-27 which is six years later than the original modification (revision 1.16 → 1.17) and in *OpenBSD* more than eight years later on 2001-

11-21 (revision 1.7 → 1.8)—though without the *(caddr_t)* type cast listed in the preceding code snippet. The *diff*-snippet below depicts the modification.

Listing 5.7 Source code change in NetBSD (seven years later).

```
< sleep (( caddr_t )dq,  PINOD+2);
– – –
> (void)  tsleep (dq,  PINOD+2, ”dqsync”, 0);
```

In the *NetBSD* variant of the change log the comment is less informative: “*sleep() -> tsleep()*”. While in *NetBSD* this change still produces similar results when building the revision deltas via *diff*, in *OpenBSD* the change was part of a larger source code modification consisting of 380 added and 161 deleted source lines (CVS does not identify modified lines, instead every modified line accounts for one added and one deleted line). Analogous to the given example, many changes can be found with varying degree of similarity making it difficult to track source code propagation.

5.7.6 Discussion

During experiments with our *Release History Database* we noticed some shortcomings which have to be resolved prior to a thorough analysis of the different product variants. First, through moving and renaming files in the CVS repository by the developers of the software systems, the historical information is segmented. Thus related segments have to be identified and concatenated to describe a continuous historical time-line of an artifacts history. Second, as result of the import process artifacts which have identical file names are assigned different IDs in the *Release History Database*. This may negatively effect multi-database queries for comparison of artifacts since artifacts with common origins have to be identified for every evaluation of a database query. This mapping of IDs will be ideally stored in the consolidated part of the *Release History Database* as indicated in Figure 4.8.

From the software evolution analysis point of view, BSD represents an interesting software system which opens a wide field for further analysis. Since detailed information about the source code is available it would be beneficial to apply a tool for code clone detection such as [84] proposed by Kamiya et al. To improve the results of the lexical search we currently explore the application of techniques related to Latent Semantic Indexing (LSI) [96].

5.8 Résumé

In the course of the implementation of this case study, we have identified a number of interesting evolutionary properties of the *Mozilla Application Suite* with our *EvoGraph* approach. All file-sets maintain a high indirect structural dependency which stems from the large number of common component interfaces they use. Moreover, S3 has also a direct structural dependency which makes things worse. As a consequence, both files in S3 can be considered to have an instable structural dependency and will exhibit co-change behavior in the future as well. With respect to the *Mozilla* application files *navigator.js* and *navigator.xul* which constitute the browser user interface, our results about their structural dependencies and their frequent modifications are not surprising. As shown by means with our *navigator.js* and *nsBrowserInstance.cpp* example, the results underpin the option offered by the *EvoGraph* approach to analyze other types of artifacts than source code—such as **.cpp* and **.h* files—as well for their structural stability. Further not only to our case study restricted results are:

- the visualizations provide comprehensive, qualitative feedback about the logical coupling between structural dependencies which not only allows to identify the responsible source code entities but also allows to identify interesting evolutionary (anti-) patterns;
- detected anti-patterns provide means to assess the success of a design and its structural stability with respect to its evolvability;

- re-engineering events can be identified and can assessed to identify successful or unsuccessful events;
- tracing source code links into user interface related files provides insight into structural dependencies between different implementation concepts which are not captured by other software analysis programs; and
- from the mining perspective, change transactions exhibit a more complex and heterogenous structure with respect to dependency exposure than expected. This increases the effort to fully track and understand the evolutionary impact of source code changes.

Though, some basic knowledge about the application itself is required, the implementation of the parsers for the fact extractors is easy to accomplish. A further refinement of the source code change analysis will facilitate the reasoning about change types, e.g., method signature changes, function body changes, or control blocks.

Comparing execution traces is a simple but efficient way to gain information about changes in the "as-implemented" architecture without the need to have access to the source files. The extracted information can be used to recover interaction patterns between different entities such as methods, files, or modules.

EvoTrace allows us to track the evolution of selected modules and present the findings in three different kinds of visualizations: Gantt diagrams, Matrix views and Kiviat diagrams. Based on these graphical representations, we have shown that certain aspects such as invocation structures between modules can be tracked and comprehended quite effectively. The properties of execution traces, such as detailed information about "scheduling" data, invocation patterns, call frequency, nesting-levels, or threading, complements results gained from release history and structural analysis.

We used *EvoTrace* to analyze and compare the execution traces of two different versions of the *Mozilla Application Suite* to obtain insights into its longitudinal development. With respect to our case study, we were able to determine the evolution from a "dispatcher" oriented communication in version 1.4 to a more direct communication between software modules in version 1.7.

Retrospective analysis of related products with *EvoFamily* opens interesting perspectives on the evolution of large software systems. With minimal changes and additions to existing tools it is already possible to recover the information flow between the different variants and evolutionary hot-spots with respect to the module structure. Through the application of a lexical search in the change logs we were able to reveal the increasing information flow of two variants of the systems. For the third system we found a decreasing flow starting from a very high level. For one selected system we applied an adapted method which generates high-level views of the module structure of a system with respect to their coupling and information flow from other product variants. To support these findings about the information flow we performed detailed change analysis of a randomly selected file. Interesting findings with respect to the inspected systems are: the wide distribution of *Linux* related change dependencies in the source code; the strong change coupling within the subtree of src/sys; and the propagation of source code taking several years.

Chapter 6

Conclusions and Future Work

In this chapter we present our conclusions and outline possible areas of future work based on the knowledge gained during the work on this thesis. Since the choice of the case study has a significant impact on the obtained results, we think that the selection of a large open source software system contributes much to transparency and comparability of different approaches for software analyses in general. The *Mozilla Application Suite*, which served us as case study for a number of research experiments, provided us with many surprising results in terms of evolutionary and structural properties.

6.1 Conclusions

EvoGraph is an efficient and lightweight approach to systematically identify evolutionary hot-spots in the longitudinal development of large scale software system. We successfully applied our prototype implementation on the *Mozilla Application Suite* and found interesting structural shortcomings which persisted over a longer period of time and still exist though the developers are aware of them.

As shown in our case study, we performed our analyses on different abstraction levels such as file-, module-, and feature-level. Interesting results on file-level were obtained in cross-language analyses such as scripting language and user interface definition representing different abstraction levels. The results showed the required communication points and the synchronous changes in the two abstraction levels. However, we performed most analyses on module-level based on the results obtained from the *stickiness*-view. Here, we were able to identify evolutionary anti-patterns such as the *day-fly* pattern. It can be interpreted as problems with the implementation or insufficient knowledge about possible side-effects. A quantitative evaluation of selected logically-coupled entities with respect to structural stability confirmed our opinion about hot-spots of the source code. On feature-level we obtained interesting results with respect to the interwovenness of the features such as fXml and $\mathrm{fMathML}$. The opposite, an example for successful re-engineered, are the dependencies between the features fHttp and fHttps which have been removed completely.

EvoTrace, our approach to exploit dynamic information for evolution analysis provides an insight into the effect of changes—also beyond method-level. The results depicted on a large scale show the shift from a central oriented communication to a more distributed one. The approach is a valuable supplement to pure source code analysis approaches.

With respect to our *EvoFamily* extension we were able to successfully evaluated the approach against a set of three related products. In the course of this evaluation we identified interesting commonalities between the systems. As a consequence of the parallel development of the systems, sometimes bug fixes took several years to propagate from one system to the other.

In the current version of our *Release History Database*—with respect to the case study—we evaluated only a small part of the available problem report information. As most of the information is available as free text, a more detailed evaluation is difficult. More machine interpretable date such as type and semantic of change are desirable extensions. Nevertheless, problem report information are valuable for instance in tracking related change reports across different time-slots as they occur in the *day-fly* pattern or

in identifying recurring problems.

EvoGraph and the related approaches are part of the *EvoZilla* framework which has proven to be a flexible platform for the implementation of our retrospective software evolution analysis prototypes. A further integration with the development process is useful and could be accomplished via its integration into an IDE such as Eclipse [1] to provide immediate feedback to the developers.

6.1.1 Accept / reject hypotheses

For the fulfillment of our research goals, we formulated four hypotheses which have been validated in the course of our case study. Following we address these hypotheses and justify their respective acceptance or rejection:

- *Correlation hypothesis H1a*: accepted.

  Through systematic filtering of change reports and enhancement with problem report information we obtained a set of logical couplings which corresponds with structural dependencies. With respect to our case study, such dependencies are frequently caused by inheritance relationships, method invocations, variable accesses of shared classes providing some base functionality. Direct dependencies of logically-coupled entities were seldom to observe. An important filter criterion was the size of the co-change transactions. The larger the number of files participating in an transaction, the higher the probability of changes with little or no effect on the functionality of the system.

- *Traceability hypothesis H1b*: accepted.

  Our observations from the case study indicate the coherence between source code metrics and the degree of logical coupling. Thus, entities having a high fan-out consequently have higher number of co-changes due to a number interface signature changes. As these shortcomings have not been remedied for a significant number of release in our test system, the frequent co-changes are therefore recorded in the systems history.

- *Stickiness filter hypothesis H2*: accepted.

  Considering our case study, we successfully applied our filtering mechanism to extract those entities violating the systems modularity. Furthermore, visual clustering revealed their detailed interdependencies without the clutter of entities having only local logical coupling. Consequently, from the depicted entities in the *stickiness*-view the design-smells such as God classes are easy to identify. Results obtained also conform with publicly available information on the case studies home page and our earlier empirical studies.

- *Source diff hypothesis H3*: accepted.

  Via the application of special parsers and source code heuristics about the system under inspection, we extracted information on a quantitative and qualitative sufficient level. The results correspond with our other empirical studies about the *Mozilla Application Suite* with respect to structural dependencies. Furthermore, based on the available evidence we detect anti-patterns such as *day-fly*, *pulsar*, or *skipjack* in the structural evolution of a system.

6.1.2 Research goals

With respect to the research goals formulated at the beginning of this thesis, we now provide our justifications based on the results and experiences obtained from our case:

- *Storage and computational model G1*: fulfilled.

  EvoZilla is the implementation of our storage and computational model enabling the longitudinal analysis of a large software system or even a family of related products. It provides the necessary data basis and infrastructure for the implementation of the subsequent analysis approaches such as *EvoGraph* or *EvoTrace*.

- *Detection of structural entities G2*: fulfilled.

  The exploitation of *stickiness* has proven to be a fast and efficient method to point out dependencies between structural relevant files. Though most changes between the files were the not surprising interface changes, they had their origin in the commonly used artifacts which we could identify with our lightweight parsers. Since the degree of exploitable coupling depends primarily on different factors such as system age, design, development method, application domain etc. we presume that the result vary significantly between systems.

- *Feedback generation G3*: fulfilled.

  As the detailed results of the case study indicate, the proposed *EvoGraph* approach delivers structural information for longitudinal analysis in sufficient quality and quantity from the source code changes. Our comparative study of structural properties has shown that a vast amount of the detailed information is unnecessary and can be neglected in the course of the evaluation of a systems structural stability. In contrast to traditional structural analysis approaches, *EvoGraph* captures the complete live-time of a project, supports the identification of longitudinal change patterns, and is efficient with respect to information extraction process.

To summarize, *EvoGraph* is a well-scalable approach to point out structural shortcomings and provides detailed structural feedback about a system's longitudinal evolution. Consequently, the indicated hot-spots are first class candidates for re-structuring or re-engineering activities whereas the generated feedback facilitates the reasoning process about the system as-implemented architecture. Another interesting application area for our approach is the monitoring of ongoing projects for evolutionary patterns. Its efficiency and scalability enable the institutionalization within an existing development environment.

6.2 Future work

During the work on this thesis we recognized a number of further research areas which are worth further exploration to extend the described *EvoZilla* framework:

- *Dynamic analysis*: Integration of execution trace information on a per release interval base for a finer-grained evolution analysis to obtain a holistic view onto the application system including configuration files. This will reveal the impact of source code changes in more detail than plain source code analysis. Challenges in the analysis are the large amount of data, efficient generation of the traces, and the identification of commonalities in the execution traces;

- *Structural information*: Currently, in our model structural dependencies are extracted and used on a low level only. Since structural information such as names of components, methods, global constants, etc., do not change frequently between different releases, a sufficient approximation—especially in the retrospective analysis process—for intermediary revisions can be made. By augmenting our model with this "interpolated" structural information the impact of structural changes can be estimated more accurately and the quality of feedback can be improved as well;

- *Fine-grained source code changes*: Related with the extraction of structural information is the provision of information about source code changes on a fine-grained level. Valuable would be the semantics of the source code change and some classification of the change according to the *IEEE standard for software maintenance*. The additional value of this information for the retrospective analysis is that finer granular information such as control constructs, variable declarations, mathematical operations are evaluated as well. It allows for instance to reason which changes have propagating effects and how they propagate through the structure. The estimated severity, e.g., new control constructs added, of the source code enhance the visualizations of *EvoGraph* and assessment of structural stability;

- *Semantic information*: The current model considers semantic information only in a very limited way such as build architecture or GUI related changes obtained from file-type analysis. Other potential sources for semantic information are the text portions of the modification reports, associated problem

reports or in-line documentation. The difficulty with these sources is the information extraction from the free-text messages composed of prose in developer jargon;

- *Code clone data*: Integration of results from code clone detection are exploitable as additional source for coupling dependencies between files. As with other data, code clones have to be filtered and classified to become valuable and usable information. In contrast to logical coupling based on the release history, code clone based coupling does not require a ripening process. It can be used instantaneously in the source code version it has been detected in;

- *Product family evolution*: Again, code clone detection between different products can be used to provide more information about artifacts and possibly related modifications. An analysis can reveal the degree of how tight product variants are coupled and which changes are to expect if a modification is applied onto one product of the product family. Another interesting area for future work is the detailed analysis of change log information for commonalities, since they may provide relevant information about the origin of propagating changes as our case study indicates.

Some of the above research questions have been briefly addressed in the course of this thesis. From our point of view the most interesting extensions to the *EvoGraph* approach is an evaluation model about architectural stability. Based on the collected information, it will be possible to show in detail how the system accommodates new requirements, what changes have been made and which parts of the architecture are negatively affected.

Appendix A

Structure and Evolution

In Chapter 4 we introduced our *EvoGraph* approach and explained its theoretical foundations. To validate the main hypothesis *H1a* we will study the structure and evolution the *Mozilla Application Suite* on a high-level view between different selected modules. The objective is to find correlations between module coupling and logical coupling.

Therefore, we developed an approach to generate higher-level views of the structure of a software system to study the coupling between structural elements. Typically, these views are depicted as graphs whereas nodes represent the structural elements and edges the relationships between them. In particular, relationships represent dependencies between structural elements that lead to *coupling* between these elements. In theory, strongly coupled structural elements are more likely to be modified together than are loosely coupled elements. Therefore, structural designs concentrate on encapsulating common behavior within an structural element, consequently increasing cohesion and lower coupling [23]. An *structural element* in the context of this chapter is a *software module* that results from the decomposition of a software system into implementation units. According to Clements et al. [37] we refer to a software module as an implementation unit of software that provides a coherent unit of functionality. Modules present a code-based way of considering the system [23].

The approach which we called *ArchEvo,* enriches source code models extracted from source code and execution traces with logical coupling data obtained from the *Release History Database.* The data sources are integrated into a common directed attributed graph from which *ArchEvo* abstracts higher-level views using architecture recovery [112]. In the analysis of the dependencies between architectural elements *ArchEvo* correlates both types of abstracted coupling relationships and shows strongly coupled elements as-implemented but also verifies these couplings by release history data. Consequently, the architectural views computed by *ArchEvo* provide an integrated view on the structure and its evolution.

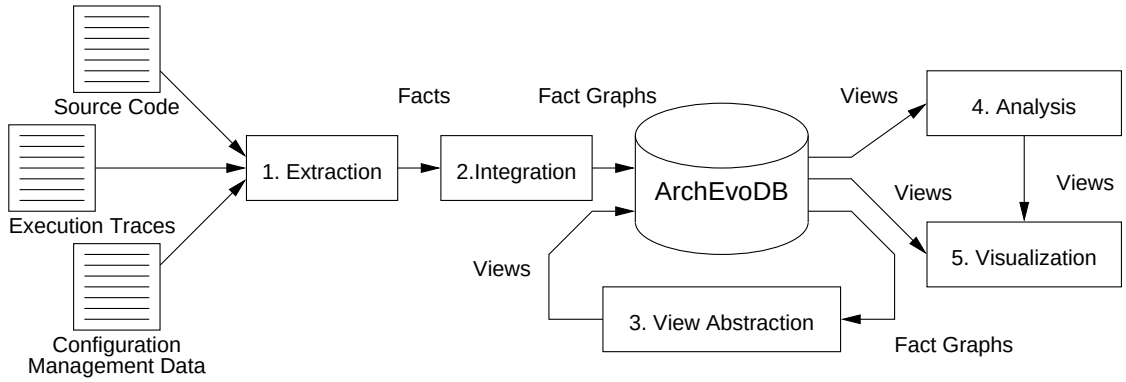
A.1 Approach

Recent research in analyzing structural elements concentrated on information obtained from source code and the running system. Briand et al. reported on the different measurements and described a framework of coupling measurements between classes and objects [32].

In the *EvoGraph* approach we focused on investigating release history information including version, change, and defect data to obtain information about logical couplings between source code entities [52] and then to deduce structural shortcomings.

The *ArchEvo* approach is a combination of both approaches mentioned before to analyzing coupling relationships on the structural level. Figure A.1 depicts the process followed by *ArchEvo.* The process steps are described in the following subsections.

Figure A.1 ArchEvo architecture evolution analysis process.



A.1.1 Fact extraction

Implementation specific data (i.e. facts) is obtained by applying static and dynamic analysis techniques including parsing and profiling. Parsing delivers static source code models that contain the source code specific entities such as files, packages, classes, methods, and attributes and the dependencies between them. Dependencies are file includes, class inherits and aggregates, method calls and overrides, and variable accesses. Profiling delivers run-time data (i.e. method call sequences) for an executed scenario and complements static source code models.

The release history information is the same as we used for the *EvoGraph* case study. Logical coupling and other relevant evolutionary information are therefore taken from our *Release History Database*. Currently, logical coupling is detected on the level of source files which is sufficient for this approach. However, the integration of logical coupling on a more fine-grained level would be beneficial and is subject to future work.

As mentioned before different tools are used to obtain facts from a software system each using its own output format. For instance, static source code models and execution traces are stored in an ASCII file as directed attributed graphs (fact graphs in Figure A.1). *ArchEvo* uses the Rigi Standard Format (RSF) [141] for storing these graphs. Nodes represent extracted source code entities (e.g. files, classes, methods, attributes) and edges represent the relationships between them (e.g. includes, inherits, invokes, accesses).

To facilitate a common access of both data sources they have to be integrated into a common repository which is the *ArchEvoDB*. Because the *ArchEvo* abstraction approach needs directed attributed graphs, the *Release History Database* data is converted to a fact graph. Nodes of this graph represent source files and modification reports and edges represent couplings between source files. Next, these fact graphs are integrated into the *ArchEvoDB* that is a fact graph containing the source code, run-time and logical coupling data.

A.1.2 Data integration

The two basic requirements for integrating the extracted heterogeneous fact graphs are: (a) facilities for extending the meta model to integrate new entity and relationship types; and (b) algorithms to map local to unique identifiers. Concerning the first requirement we use the FAMIX meta model for object-oriented programming languages [123] and extend it towards the inclusion of configuration management data, architectural views and metric data. Latter data is computed by the *ArchEvo* view abstraction algorithm described in the next subsection. The second requirement is fulfilled by our integration tool that maps locally unique identifiers (within a data file) to identifiers unique within the repository (*ArchEvoDB*).

Each fact graph is read by the integration tool that for each entity and relationship contained in the fact graph finds out about its identifier in the repository and if not exists computes a new one. Using the unique identifiers the new facts (nodes, edges, and attribute records) are added to the *ArchEvoDB*. The result is a common repository containing the integrated fact graph that forms the basis for the on-going abstraction

Table A.1 Measures computed for relationships abstracted to the module-level.

Measurement	Description
nrRelsDirect	# of abstracted direct lower-level relationships
nrRelsIndirect	# of abstracted indirect lower-level relationships
nrAdirectB	# of source code entities of direct relationships in module A
nrAindirectB	# of source code entities of indirect relationships in module A
nrBdirectByA	# of source code entities of direct relationships in module B
nrBindirectByA	# of source code entities of indirect relationships in module B
refcount	# of modification reports of a logical coupling relationship

and analysis tasks.

A.1.3 View abstraction

In this step architectural views are abstracted from the integrated fact graph. *ArchEvo* supports abstraction to different levels of abstraction whereas the level is specified by the user. The abstraction algorithm used by *ArchEvoDB* is based on the approach presented by Holt et al. in [78], but extends it by computing measures for abstracted elements and relationships. An approach similar to Holt's also has been described by Feijs et al. in [50].

Relationships between architectural elements and abstraction measures are computed using binary relational algebra. Currently, we use the grok tool [49] for calculating the binary relations because grok is able to handle extracted and integrated fact graphs in RSF format. However, the abstraction of attributes of relationships is not straight forward with grok, hence we implemented a workaround to handle this problem: For instance, the attribute values of lower-level relationships that form an abstracted relationship are summed up. Ongoing work is concerned with storing fact graphs in a relational database and use the standard query language SQL instead of grok.

Algorithm 4 defines the *ArchEvoDB* abstraction algorithm that is applied to the directed attributed fact graph.

Algorithm 4 ArchEvo abstraction algorithm.

1: **for all** $entitypair(A, B)$ **do**
2: $setA \leftarrow$ entities contained by A
3: $setB \leftarrow$ entities contained by B
4: $relsAB \leftarrow$ relationships of type T between $setA$ and $setB$
5: **if** $\#relsAB > 0$ **then**
6: $rel \leftarrow$ create relationship of type T between A and B
7: $measures \leftarrow$ compute abstraction measures of rel
8: **end if**
9: **end for**

Having selected a relationship type to be abstracted the algorithm processes each pair of higher-level entities and first computes the two sets of entities (e.g. methods) contained in A and B (line 2,3). Next, the relationships of type T between the entities of set A and set B are queried (line 4) from the graph. If there is at least one relationship between any two lower-level entities of set A and B then an *abstracted* relationship between A and B is established (line 6). Measures concerning the number of affected lower-level relationships and entities are computed and stored in attributes of the new relationship (line 7). For instance, the number of modification reports making up a logical coupling is summed up and stored in the *refcount* attribute of the abstracted coupling relationship. Table A.1 lists the measures computed for abstracted relationships (these measures also apply to other levels of abstraction).

Basically, *ArchEvo* distinguishes between direct and indirect dependencies whereas indirect stands for transitive. For both kinds of dependencies the number of involved source code entities are computed.

Resulting measures reflect the weight of abstracted relationships and consequently quantify the coupling between architectural elements. They are used in the analysis of the dependencies between architectural elements.

A.1.4 Analysis

The goal of the analysis step is to indicate strongly coupled elements and to provide clues why these elements have such a strong coupling. The data used for this analysis is stored in the abstracted views. They contain the architectural elements (nodes), the coupling relationships (edges), and the coupling measures (attributes).

Coupling measures are stored in attributes of (abstracted) relationships. For instance, the number of method calls is stored in the *nrRelsDirect* attribute of an abstracted *invokes* relationship. For an abstracted *couples* relationship the number of modification reports is stored in the *refcount* attribute. Based on these attributes *ArchEvo* uses graph queries to determine the relationships of interest and the corresponding architectural elements.

Graph queries are implemented using a combination of binary relational algebra and Perl scripts. For example, to determine the elements with the strongest logical coupling *ArchEvo* applies a query to the *refcount* attribute of *couples* relationships that have a value greater than a given threshold.

For the correlation of source code coupling with logical coupling relationships *ArchEvo* ranks each relationship with respect to the computed average or maximum of a given relationship attribute (e.g. *refcount*). The ranking values are represented in matrices one per attribute. Using statistical methods on the matrices the correlation between the different relationships is computed providing users with quantitative measures about the dependencies.

The result of the quantitative analysis are refined architectural views that facilitate an assessment of the current architecture and its evolution, as well as the identification of design shortcomings. They also provide good starting points for a more detailed analysis of architectural dependencies, for instance, by selecting two modules that are strongly coupled.

The detailed analysis that qualifies and verifies quantitative measures is performed on a finer-grained level of abstraction such as the file-level. Considering the reduced set of files of the selected higher-level entities (i.e. modules) the logical coupling relationships are qualified with respect to the source code coupling that caused it. Next the results of the qualification are reflected back to the higher-level views to enrich them with more details. They direct to locations of design shortcomings that should be resolved to smoothen evolution and maintenance.

The result of *ArchEvo* comprises a set of refined architectural views that show source code and logical coupling dependencies on different levels of abstraction. Higher-level views on software modules or features contain quantified coarse-grained information about elements and their relationships. In the next section we study the *Mozilla Application Suite* for correlations between structural and logical couplings.

A.2 Investigating the coupling within Mozilla

The outcome of the *ArchEvo* structural analysis process are views that indicate the correlation between structural coupling of files or modules as the cause and logical coupling as effect. To extract the facts for this study, we applied our *ArchEvo* approach onto version 1.3a of the *Mozilla Application Suite*. Starting from the design documentation we focused our analyses on a selected set of software modules as architectural elements that implement the internal representation (i.e. content) and the layout of web pages. Table 5.12 lists the selected software modules together with corresponding source code directories containing their implementation. The mapping between modules and source code directories has been taken from *Mozilla*'s design documentation.

Subgoals in our analyses were: (a) abstraction from the low-level information to the level of software modules; and (b) correlating the abstracted implementation specific relationships with the modification specific ones. The objective was to obtain measurements (sizes, weights) of different coupling dependencies between the selected software modules including source code but also logical couplings as listed in Table A.1

Based on these views and measurements we analyzed the as-implemented design of these modules with respect to their maintainability and evolvability. These two related quality attributes of software systems are influenced by the coupling between software modules. Basically, the stronger the coupling is the more effort has to be spent for maintaining and evolving the system [32]. The following sections report on our findings about the selected modules listed in Table 5.12.

A.2.1 Module view

The module view reflects the as-implemented design together with the release history information. The elements of the representation are software modules, their source code and logical coupling relationships.

The resulting graphs—different types of relationships can be selected for the graph generation—gives a first quantitative feedback about inter-module coupling. Figure A.2(e) depicts invocations—represented as red/solid arcs—between the selected modules which are represented as gray boxes. Width and height of the boxes indicate the size of software modules in terms of number of global functions and methods (width) and global variables and attributes (height) of a module. The distance between two modules is determined by the number of logical couplings between these modules and indicated as straight, cyan/solid line. The values for the logical couplings stem from filtered information of our *Release History Database*.

With this view one can easily spot the strong coupling between the three modules in the center of the graph (New Layout Engine, DOM, XPToolkit). Interesting to see is the high number of mutual calls between these modules. Consequently, when modifying one of these modules it is very likely that the other modules have to be touched.

The other graphs in Figure A.2 show the same coupling graph with different structural properties. For instance Figure A.2(d) depicts the inheritance relationship as obtained from the fact extraction. The strongest edges indicate a high correlation with the strong couplings between the modules. In both views the mutual dependencies can be observed as well. As a result, abstracted module views pointed out locations of strong couplings that caused pairwise modifications of software modules.

A.2.2 Detailed module view

For a detailed evaluation of the coupling between software modules we selected the modules New Layout Engine and XPToolkit. The focus was on the source files of both modules that have the strongest coupling. These files represent the design critical source code entities. The resulting graph comprises six files and is depicted in Figure A.3. It shows method invocations (red/solid arcs) gathered from the runtime data and the logical couplings between source files (straight cyan/solid lines).

The layout, i.e., the relative position of the boxes to each other, is defined by the number of logical couplings found between files. Actually, the highest coupling crossing the module boundaries exists between *nsPresShell.cpp* and *nsXULDocument.cpp* with 81 problem reports. The flags with the numbers of invocations attached to the arcs are always pointing from the caller and indicate the actual number of dynamic invokes found. For example, there are four calls from *nsXULElement.cpp* to *nsPresShell.cpp* and three calls in the other direction.

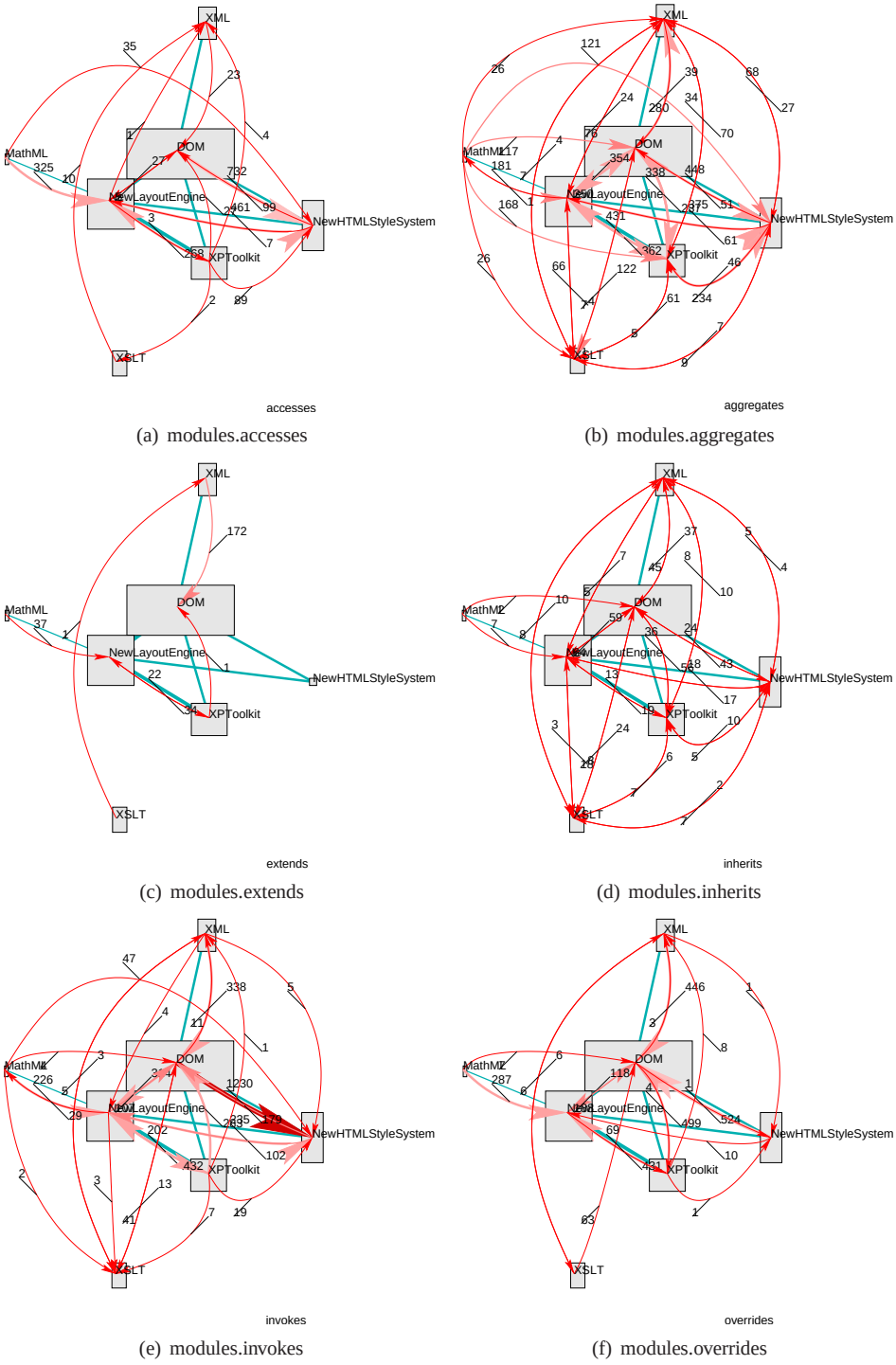
The central position of *nsCSSFrameConstructor.cpp* indicates a high degree of coupling with other files. This strong logical coupling is further strengthened by the method invoke relationships which cover all other files in this view. Therefore, this file is the most critical entity concerning evolving or maintaining the two modules.

Summarized, the case study showed the bottom-up abstraction of lower-level information to architectural views (i.e. module view). These views are mandatory to point out the modules that are most involved in pairwise changes. Next going top-down from architectural views to lower-level views the details making up and causing these logical couplings in the implementation are revealed.

A.3 Résumé

The *ArchEvo* approach combines information gained from static and dynamic analyses of the source code with release history data into specific views on different abstraction levels. The analysis applies fact ex-

Figure A.2 Contrasting the results for different structural coupling: accesses, aggregates, extends, inherits, invokes, and overrides.



(a) modules.accesses

(b) modules.aggregates

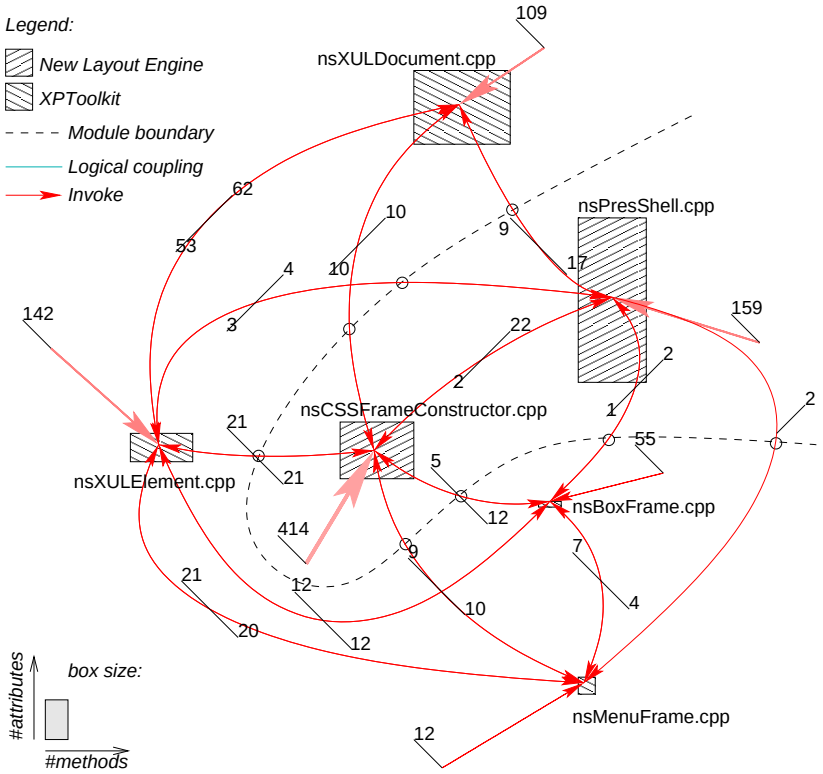
(c) modules.extends

(d) modules.inherits

(e) modules.invokes

(f) modules.overrides

Figure A.3 Invocations between files (Modules XPToolkit and New Layout Engine).



traction and generates specific directed attributed graphs; nodes represent source code entities and edges represent relationships such as accesses, includes, inherits, invokes, and coupling between certain architectural elements. The integration of data is then performed on a meta model level to enable the generation of structural views using binary relational algebra. These integrated structural views show intended and unintended couplings between structural elements, hence pointing point out locations in the system with structural and logical coupling.

Appendix B

File Sets

Table B.1: File-sets with path evaluated in the case study.

Set	subset	cvsitemid	Module	File
S1	A	2423	*content/base/src*	*nsRuleNode.cpp*
S1	A	2754	*content/shared/public*	*nsStyleStruct.h*
S1	A	2782	*content/shared/src*	*nsStyleStruct.cpp*
S1	A	2770	*content/shared/src*	*nsCSSProps.cpp*
S1	A	2548	*content/html/content/src*	*nsHTMLTextAreaElement.cpp*
S1	A	2619	*content/html/style/src*	*nsCSSDeclaration.cpp*
S1	A	2629	*content/html/style/src*	*nsCSSParser.cpp*
S1	A	2640	*content/html/style/src*	*nsCSSStyleRule.cpp*
S1	A	2658	*content/html/style/src*	*nsHTMLCSSStyleSheet.cpp*
S1	A	2946	*content/xbl/src*	*nsBindingManager.cpp*
S1	A	2999	*content/xbl/src*	*nsXBLService.cpp*
S1	A	2947	*content/xbl/src*	*nsXBLBinding.cpp*
S1	A	3151	*content/xul/content/src*	*nsXULElement.h*
S1	A	4632	*dom/src/base*	*nsDOMClassInfo.cpp*
S1	B	14170	*layout/html/style/src*	*nsCSSFrameConstructor.cpp*
S2	A	10703	*htmlparser/src*	*nsParser.cpp*
S2	A	10681	*htmlparser/src*	*nsHTMLTokenizer.cpp*
S2	A	10623	*htmlparser/src*	*CNavDTD.cpp*
S2	A	10654	*htmlparser/src*	*nsElementTable.cpp*
S2	B	2575	*content/html/document/src*	*nsHTMLContentSink.cpp*
S3	A	3187	*content/xul/document/src*	*nsXULDocument.cpp*
S3	B	13937	*layout/html/base/src*	*nsPresShell.cpp*
S4	A	6901	*extensions/cookie*	*nsCookieService.cpp*
S4	A	6903	*extensions/cookie*	*nsCookies.cpp*
S4	A	6928	*extensions/cookie*	*nsPermissions.cpp*
S4	A	6932	*extensions/cookie*	*nsUtils.cpp*
S4	A	22658	*netwerk/base/src*	*nsSocketTransport.cpp*
S4	A	22659	*netwerk/base/src*	*nsSocketTransport.h*
S4	A	22663	*netwerk/base/src*	*nsSocketTransportService.h*
S4	A	22677	*netwerk/base/src*	*nsStorageTransport.h*

(continued on next page)

(continuation from previous page)

Set	subset	cvsitemid	Module	File
S4	A	22680	*netwerk/base/src*	*nsStreamListenerTee.cpp*
S4	A	22681	*netwerk/base/src*	*nsStreamListenerTee.h*
S4	A	22771	*netwerk/cache/src*	*nsCacheMetaData.cpp*
S4	A	23082	*netwerk/protocol/http/src*	*nsHttp.cpp*
S4	A	23085	*netwerk/protocol/http/src*	*nsHttpAuthCache.cpp*
S4	A	23095	*netwerk/protocol/http/src*	*nsHttpConnection.cpp*
S4	A	23096	*netwerk/protocol/http/src*	*nsHttpConnection.h*
S4	A	23098	*netwerk/protocol/http/src*	*nsHttpConnectionInfo.h*
S4	A	23106	*netwerk/protocol/http/src*	*nsHttpHeaderArray.cpp*
S4	A	23107	*netwerk/protocol/http/src*	*nsHttpHeaderArray.h*
S4	A	23111	*netwerk/protocol/http/src*	*nsHttpPipeline.cpp*
S4	A	23117	*netwerk/protocol/http/src*	*nsHttpRequestHead.cpp*
S4	A	23118	*netwerk/protocol/http/src*	*nsHttpRequestHead.h*
S4	A	23119	*netwerk/protocol/http/src*	*nsHttpResponseHead.cpp*
S4	A	23120	*netwerk/protocol/http/src*	*nsHttpResponseHead.h*
S4	A	23121	*netwerk/protocol/http/src*	*nsHttpTransaction.cpp*
S4	A	23122	*netwerk/protocol/http/src*	*nsHttpTransaction.h*
S4	A	23244	*netwerk/socket/base*	*nsSocketProviderService.cpp*
S4	A	34051	*xpcom/ds*	*nsTime.h*
S4	A	34301	*xpcom/proxy/public*	*nsProxyRelease.h*
S4	B	23242	*netwerk/socket/base*	*nsSOCKSSocketProvider.cpp*
S4	B	26080	*security/manager/boot/src*	*nsSecurityWarningDialogs.cpp*
S4	B	26307	*security/manager/ssl/src*	*nsNSSCallbacks.cpp*
S4	B	26318	*security/manager/ssl/src*	*nsNSSCertificate.cpp*
S4	B	26326	*security/manager/ssl/src*	*nsNSSIOLayer.cpp*
S4	B	26345	*security/manager/ssl/src*	*nsSSLSocketProvider.cpp*
S5	A	35330	*xpfe/browser/resources/content*	*navigator.js*
S5	B	35331	*xpfe/browser/resources/content*	*navigator.xul*

Appendix C

Release Dates

Table C.1: Mozilla official releases, timestamps extracted from the respective source code packages and auto-generated release information based on revision tags from CVS.

ID	Release	Release date	File timestamp	Revision date
53	Mozilla 1.7.1	2004-07-08 00:00:00	2004-07-08 21:16:39	2004-07-08 00:00:00.0
52	Mozilla 1.7	2004-06-17 00:00:00	2004-06-18 00:00:24	2004-06-17 00:00:00.0
51	Mozilla 1.7 Beta	2004-03-18 00:00:00	2004-03-18 17:37:54	2004-03-18 08:41:46.0
50	Mozilla 1.7 Alpha	2004-02-23 00:00:00	2004-02-20 03:34:35	2004-02-19 02:15:37.0
49	Mozilla 1.6	2004-01-15 00:00:00	2004-01-16 11:46:08	2004-01-16 05:30:53.0
48	Mozilla 1.6 Beta	2003-12-09 00:00:00	2003-12-09 11:08:46	2003-12-09 08:39:22.0
47	Mozilla 1.6 Alpha	2003-10-31 00:00:00	2003-10-31 19:07:59	2003-10-29 11:53:43.0
46	Mozilla 1.5	2003-10-15 00:00:00	2003-10-15 22:39:08	2003-10-15 10:18:59.0
45	Mozilla 1.5 Beta	2003-08-27 00:00:00	2003-08-28 00:11:16	2003-08-27 06:00:15.0
44	Mozilla 1.5 Alpha	2003-07-22 00:00:00	2003-07-22 21:20:48	2003-07-11 12:34:16.0
43	Mozilla 1.4	2003-06-30 00:00:00	2003-07-01 00:42:28	2003-06-12 02:08:49.0
42	Mozilla 1.4 Beta	2003-05-07 00:00:00	2003-05-08 21:22:04	2003-05-08 04:43:08.0
41	Mozilla 1.4 Alpha	2003-04-01 00:00:00	2003-04-02 04:54:13	2003-04-01 08:23:00.0
40	Mozilla 1.3	2003-03-13 00:00:00	2003-03-13 16:56:07	2003-02-21 11:17:16.0
39	Mozilla 1.3 Beta	2003-02-10 00:00:00	2003-02-11 00:27:52	2003-02-10 10:49:42.0
38	Mozilla 1.3 Alpha	2002-12-13 00:00:00	2002-12-13 20:16:50	2002-12-12 11:17:52.0
37	Mozilla 1.2	2002-11-26 00:00:00	2002-12-01 00:25:55	2002-11-05 02:49:52.0
36	Mozilla 1.2 Beta	2002-10-16 00:00:00	2002-10-18 18:28:47	2002-10-17 05:28:53.0
35	Mozilla 1.2 Alpha	2002-09-11 00:00:00	2002-09-12 20:56:19	2002-09-12 04:36:38.0
34	Mozilla 1.1	2002-08-26 00:00:00	2002-08-27 23:59:20	2002-08-05 05:37:49.0
33	Mozilla 1.1 Beta	2002-07-22 00:00:00	2002-07-23 20:59:45	2002-07-23 12:03:02.0
32	Mozilla 1.1 Alpha	2002-06-11 00:00:00	2002-06-12 20:39:18	2002-06-11 07:41:36.0
31	Mozilla 1.0	2002-06-05 00:00:00	2002-05-31 21:17:45	2002-06-06 11:37:14.0
30	Mozilla 0.9.9	2002-03-11 00:00:00	2002-03-11 23:22:33	2002-03-11 08:18:08.0
29	Mozilla 0.9.8	2002-02-04 00:00:00	2002-02-05 00:59:14	2002-01-23 03:07:15.0
28	Mozilla 0.9.7	2001-12-21 00:00:00	2001-12-22 03:26:04	2001-12-14 09:05:35.0
27	Mozilla 0.9.6	2001-11-20 00:00:00	2001-11-21 08:33:57	2001-11-09 04:18:56.0
26	Mozilla 0.9.5	2001-10-12 00:00:00	2001-10-15 23:33:21	2001-10-05 08:41:04.0
25	Mozilla 0.9.4	2001-09-14 00:00:00	2001-09-15 00:17:44	2001-09-15 01:18:52.0

(continued on next page)

(continuation from previous page)

ID	Release	Release date	File timestamp	Revision date
24	Mozilla 0.9.3	2001-08-02 00:00:00	2001-08-03 23:42:05	2001-07-31 05:10:29.0
23	Mozilla 0.9.2	2001-06-28 00:00:00	2001-06-30 03:02:08	2001-06-30 01:54:34.0
22	Mozilla 0.9.1	2001-06-07 00:00:00	2001-06-18 22:23:43	2001-05-31 02:25:33.0
21	Mozilla 0.9	2001-05-07 00:00:00	2001-05-08 01:33:27	2001-04-25 04:24:24.0
20	Mozilla 0.8.1	2001-03-26 00:00:00	2001-03-27 18:47:42	2001-03-17 01:39:04.0
19	Mozilla 0.8	2001-02-14 00:00:00	2001-02-19 20:18:23	2001-02-10 01:05:43.0
18	Mozilla 0.7	2001-01-09 00:00:00	2001-01-10 20:19:08	2000-12-27 12:08:18.0
17	Mozilla 0.6	2000-12-06 00:00:00	2000-12-13 03:09:42	2000-11-14 10:24:43.0
16	Milestone 18	2000-10-12 00:00:00	2000-10-12 21:36:50	2000-10-05 02:47:22.0
15	Milestone 17	2000-08-07 00:00:00	2000-08-08 21:59:20	2000-07-27 06:25:31.0
14	Milestone 16	2000-06-13 00:00:00	2000-06-15 03:56:45	2000-06-05 03:19:32.0
13	Milestone 15	2000-04-18 00:00:00	2000-04-21 18:55:03	2000-04-11 06:05:28.0
12	Milestone 14	2000-03-01 00:00:00	2000-03-03 19:46:03	2000-03-01 01:21:42.0
11	Milestone 13	2000-01-26 00:00:00	2000-01-27 04:14:15	2000-01-21 03:08:55.0
10	Milestone 12	1999-12-21 00:00:00	1999-12-22 01:28:34	1999-12-19 07:59:19.0
9	Milestone 11	1999-11-16 00:00:00	1999-11-17 03:13:20	1999-11-09 09:31:43.0
8	Milestone 10	1999-10-08 00:00:00	1999-10-09 04:53:07	1999-09-29 04:46:49.0
7	Milestone 9	1999-08-26 00:00:00	1999-08-26 19:33:49	1999-08-17 02:03:26.0
6	Milestone 8	1999-07-16 00:00:00	1999-07-16 00:54:27	1999-07-13 04:19:17.0
5	Milestone 7	1999-06-22 00:00:00	1999-06-23 05:16:46	1999-06-19 02:38:19.0
4	Milestone 6	1999-05-29 00:00:00	1999-05-29 01:37:21	1999-05-24 11:35:06.0
3	Milestone 5	1999-05-05 00:00:00	1999-05-12 00:24:36	1999-05-03 11:52:12.0
2	Milestone 4	1999-04-15 00:00:00	1999-04-16 22:00:17	1999-04-28 12:42:28.0
1	Milestone 3	1999-03-19 00:00:00	1999-03-19 23:30:19	1999-03-17 06:17:24.0

Appendix D

Selected Publications

Following, we provide a list of the selected publications with title and proceedings they were published in. This overview is organized with respect to different research fields.

D.1 Papers and journal papers

D.1.1 Release history

- [56] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proceedings International Conference on Software Maintenance (ICSM'03)*, pages 23–32, September 2003.

D.1.2 Clustering

- [57] Michael Fischer, Martin Pinzger, and Harald Gall. Analyzing and Relating Bug Report Data for Feature Tracking. In *10th Working Conference on Reverse Engineering (WCRE)*, pages 90–99, November 2003.
- [52] Michael Fischer and Harald Gall. Visualizing Feature Evolution of Large-Scale Software based on Problem and Modification Report Data. *Journal of Software Maintenance and Evolution*, 16(6):385–403, November/December 2004.

D.1.3 Evolution and structure

- [112] Martin Pinzger, Harald Gall, and Michael Fischer. Towards an integrated view on architecture and its evolution. In *Proceedings of the Software Evolution through Transformations: Model-based vs. Implementation-level Solutions (SETra'04)*. Elsevier Science Publishers: Utrecht, Netherlands, 2004.
- [118] Jacek Ratzinger, Michael Fischer, and Harald Gall. Improving evolvability through refactoring. In *MSR 2005 Workshop Proceedings (MSR'05)*. ACM Digital Library, May 2005.
- [53] Michael Fischer and Harald Gall. EvoGraph: A Lightweight Approach to Evolutionary and Structural Analysis of Large Software Systems. In *13th Working Conference on Reverse Engineering (WCRE)*, October 2006.

D.1.4 Product families

- [55] Michael Fischer, Johann Oberleitner, Jacek Ratzinger, and Harald Gall. Mining Evolution Data of a Product Family. In *MSR 2005 Workshop Proceedings (MSR'05)*. ACM Digital Library, May 2005.

D.1.5 Dynamic analysis

- [54] Michael Fischer, Johann Oberleitner, Harald Gall and Thomas Gschwind. System Evolution Tracking through Execution Trace Analysis. In *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC'05)*, May 2005.

D.2 Technical reports

D.2.1 Release history

- [58] Michael Fischer, Martin Pinzger, and Harald Gall. Observing the Evolution of Modification Complexity of Logically Coupled Modules. In *Technical Report TUV-1841-2004-01*, Distributed Systems Group, Technical University of Vienna, January 2004.

Bibliography

[1] Eclipse. *http://www.eclipse.org/* [30 June 2006].

[2] Hybernate. *http://www.hibernate.org/* [30 June 2006].

[3] Imagix 4D. *http://www.imagix.com/* [30 June 2006].

[4] MySQL. *http://www.mysql.com/* [30 June 2006].

[5] Rational ClearCase. *http://www.ibm.com/software/awdtools/clearcase/* [30 June 2006].

[6] The Mozilla Bug Database. *http://bugzilla.mozilla.org/* [30 June 2006].

[7] *Merriam Webster's Collegiate Dictionary*. Merriam-Webster, Incorporated, 10th edition edition, 1996.

[8] GNU's Not Unix!, 1996–2004. *http://www.gnu.org* [30 June 2006].

[9] The GNU Bug Tracking System, 1996–2006. *http://www.gnu.org/software/gnats/* [30 June 2006].

[10] Architectural Reasoning for Embedded Systems, 1997. *http://www.itea-office.org/software_product_families_in_europe_the_esaps__caf_projects* [30 June 2006].

[11] BitKeeper, 1997–2006. *http://www.bitkeeper.com/* [30 June 2006].

[12] *IEEE standard for software maintenance*, IEEE Std 1219-1998. The Institute of Electrical and Electronics Engineers, 1998.

[13] Bugzilla Bug Tracking System, 1998–2006. *http://www.bugzilla.org/* [30 June 2006].

[14] Engineering Software Architectures, Processes and Platforms for System Families, 1999. *http://www.itea-office.org/public/project_leaflets/ESAPS_results_oct-02.pdf* [30 June 2006].

[15] Subversion - Version Control Rethought, 2000–2006. *http://subversion.tigris.org/* [30 June 2006].

[16] From Concept to Application in System-Family Engineering, 2001. *http://www.extra.research.philips.com/euprojects/cafe/* [30 June 2006].

[17] FAct-based Maturity through Institutionalisation Lessons-learned and Involved Exploration of System-family engineering, 2003. *http://www.itea-office.org/public/project_leaflets/FAMILIES_profile_oct-03.pdf* [30 June 2006].

[18] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo. An Automatic Approach to identify Class Evolution Discontinuities. In *IWPSE '04: Proceedings of the Principles of Software Evolution, 7th International Workshop on (IWPSE'04)*, pages 31–40, Washington, DC, USA, 2004. IEEE Computer Society.

[19] Giuliano Antoniol, Vincenzo Fabio Rollo, and Gabriele Venturi. Detecting groups of co-changing files in CVS repositories. In *IWPSE '05: Proceedings of the Eighth International Workshop on Principles of Software Evolution*, pages 23–32. IEEE Computer Society, 2005.

[20] Atlassian Software Systems Pty Ltd. Bug Tracking, Issue Tracking, & Project Management. *http://www.atlassian.com/software/jira/* [30 June 2006].

[21] Wayne A. Babich. *Software Configuration Management: Coordination for Team Productivity*. Addison Wesley Longman, 1986.

[22] Thomas Ball and Stephen G. Eick. Software Visualization in the Large. *IEEE Computer*, 29(4):33–43, 1996.

[23] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 2nd edition, 2003.

[24] Jennifer Bevan and Sung Kim. Kenyon: A Common Software Stratigraphy Platform. *http://www.sei.cmu.edu/cmmi/general/general.html*.

[25] Jennifer Bevan, E. James Whitehead, Jr., Sunghun Kim, and Michael Godfrey. Facilitating Software Evolution Research with Kenyon. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 177–186, New York, NY, USA, 2005. ACM Press.

[26] Dirk Beyer and Ahmed E. Hassan. Evolution Storyboards: Visualization of Software Structure Dynamics. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 248–251. IEEE Computer Society, June 2006.

[27] Dirk Beyer and Andreas Noack. Clustering Software Artifacts Based on Frequent Common Changes. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05)*, pages 259–268. IEEE Computer Society, May 2005.

[28] J. Bieman, A. Andrews, and H. Yang. Understanding Change-Proneness in OO Software through Visualization. In *Proceedings of 11th International Workshop on Program Comprehension (IWPC)*. IEEE Computer Society Press, May 2003.

[29] Ted J. Biggerstaff. Design Recovery for Maintenance and Reuse. *Computer*, 22(7):36–49, 1989.

[30] Garrett D. Birkhoff. *Lattice Theory*. American Mathematical Society, Providence, RI, USA, 3rd edition, 1967.

[31] Lionel Briand, Prem Devanbu, and Walcelio Melo. An investigation into coupling measures for C++. In *Proceedings of the 19th international conference on Software engineering*, pages 412–421. ACM Press, 1997.

[32] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[33] Andreas Buja, Deborah F. Swayne, Michael Littman, Nathaniel Dean, and Heike Hofmann. XGvis: Interactive Data Visualization with Multidimensional Scaling. pages 1–34, 2001. *http://www.research.att.com/areas/stat/xgobi/papers/xgvis.pdf* [5 July 2004], *http://www-stat.wharton.upenn.edu/~buja/PAPERS/paper-mds-jcgs.pdf* [5 July 2004].

[34] G. Casazza, G. Antoniol, U. Villano, E. Merlo, and M. Di Penta. Identifying Clones in the Linux Kernel. In *Proceedings of First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 90–97. IEEE Computer Society, 2001.

[35] Per Cederqvist, Dick Grune, Brian Berliner, Jeff Polk, and Jim Klingmon. *Version Management with CVS*, 1992. *http://www.cvshome.org/docs/manual/* [30 June 2006].

[36] Neville Churcher, Warwick Irwin, and Ron Kriz. Visualising Class Cohesion with Virtual Worlds. In *CRPITS '24: Proceedings of the Australian symposium on Information visualisation*, pages 89–97. Australian Computer Society, Inc., 2003.

[37] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.

[38] Christian Collberg, Stephen Kobourov, Jasvir Nagra, Jacob Pitts, and Kevin Wampler. A system for Graph-Based Visualization of the Evolution of Software. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 77–ff. ACM Press, 2003.

[39] Susan Dart. Concepts in Configuration Management Systems. In *Proceedings of the 3rd international workshop on Software configuration management*, pages 1–18, New York, NY, USA, 1991. ACM Press.

[40] Alan Mark Davis. *201 Principles of Software Development*. McGraw-Hill, Inc., New York, NY, USA, 1995.

[41] Serge Demeyer, Stèphane Ducasse, and Oscar Nierstrasz. *Object-oriented Reengineering Patterns*. Morgan Kaufmann Publishers, An Imprint of Elsevier Scienence: San Francisco CA, USA, July 2002.

[42] William Dickinson, David Leon, and Andy Podgurski. Pursuing Failure: The Distribution of Program Failures in a Profile Space. In *Proceedings of the Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*, pages 246–255. ACM Press, September 2001.

[43] Dirk Draheim and Lukasz Pekacki. Process-Centric Analytical Processing of Version Control Data. In *Proceedings Sixth International Workshop on Principles of Software Evolution (IWPSE'03)*, pages 131–136. IEEE Computer Society Press, September 2003.

[44] Jürgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. GUPRO - Generic Understanding of Programs. volume 72. Elsevier Science Publishers: Utrecht, Netherlands, October 2002. *http://www.elsevier.com/gej-ng/31/29/23/127/46/show/Products/notes/index.htt* [5 July 2004].

[45] S. Eick, J. Steffen, and E. Sumner. Seesoft–A Tool For Visualizing Line Oriented Software Statistics, 1992.

[46] Stephen G. Eick, Todd L. Graves, Alan F. Karr, Audris Mockus, and Paul Schuster. Visualizing Software Changes. *Software Engineering*, 28(4):396–412, 2002.

[47] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Aiding Program Comprehension by Static and Dynamic Feature Analysis. In *Proceedings IEEE International Conference on Software Maintenance (ICSM'01)*, pages 602–611. IEEE Computer Society Press, November 2001.

[48] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[49] Hoda Fahmy and Richard C. Holt. Software Architecture Transformations. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 88, Washington, DC, USA, 2000. IEEE Computer Society.

[50] L. Feijs, R. Krikhaar, and R. Van Ommering. A Relational Approach to Support Software Architecture Analysis. *Softw. Pract. Exper.*, 28(4):371–400, 1998.

[51] Patrick Finnigan, Richard C. Holt, Ivan Kallas, Scott Kerr, Kostas Kontogiannis, Hausi A. Müller, John Mylopoulos, Stephen G. Perelgut, Martin Stanley, and Kenny Wong. The Software Bookshelf. *IBM Systems Journal*, 36(4):564–593, November 1997.

[52] Michael Fischer and Harald Gall. Visualizing Feature Evolution of Large-Scale Software based on Problem and Modification Report Data. *Journal of Software Maintenance and Evolution*, 16(6):385–403, November/December 2004.

[53] Michael Fischer and Harald Gall. EvoGraph: A Lightweight Approach to Evolutionary and Structural Analysis of Large Software Systems. In *13th Working Conference on Reverse Engineering (WCRE)*, October 2006.

[54] Michael Fischer, Johann Oberleitner, Harald Gall, and Thomas Gschwind. System Evolution Tracking through Execution Trace Analysis. In *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC'05)*. IEEE Computer Society Press, May 2005.

[55] Michael Fischer, Johann Oberleitner, Jacek Ratzinger, and Harald Gall. Mining Evolution Data of a Product Family. In *MSR 2005 Workshop Proceedings (MSR'05)*. ACM Digital Library, May 2005.

[56] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proceedings International Conference on Software Maintenance (ICSM'03)*, pages 23–32, September 2003.

[57] Michael Fischer, Marting Pinzger, and Harald Gall. Analyzing and Relating Bug Report Data for Feature Tracking. In *10th Working Conference on Reverse Engineering (WCRE)*, pages 90–99, November 2003.

[58] Michael Fischer, Marting Pinzger, and Harald Gall. Observing the Evolution of Modification Complexity of Logically Coupled Modules. Technical report, Distributed Systems Group, Technical University of Vienna, 2004.

[59] Beat Fluri, Harald C. Gall, and Martin Pinzger. Fine-Grained Analysis of Change Couplings. In *Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 66–74. IEEE Computer Society, October 2005.

[60] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[61] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software Practice & Experience*, 21(11):1129–1164, 1991.

[62] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of Logical Coupling Based on Product Release History. In *Proceedings International Conference on Software Maintenance*, pages 190–198. IEEE Computer Society Press, March 1998.

[63] Harald Gall, Mehdi Jazayeri, René Klösch, and Georg Trausmuth. Software Evolution Observations Based on Product Release History. In *Proceedings of the International Conference on Software Maintenance (ICSM'97)*, pages 160–166. IEEE Computer Society Press, October 1997.

[64] Harald Gall, Mehdi Jazayeri, and Claudio Riva. Visualizing Software Release Histories: The Use of Color and Third Dimension. In *Proceedings IEEE International Conference on Software Maintenance*, pages 99–108. IEEE Computer Society Press, August 1999.

[65] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[66] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis*. Springer, 1999.

[67] Daniel M. German. An Empirical Study of Fine-Grained Software Modifications. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 316–325, Washington, DC, USA, 2004. IEEE Computer Society.

[68] Daniel M. Germán, Abram Hindle, and Norman Jordan. Visualizing the evolution of software using softChange. In *Proceedings of the 16th Internation Conference on Software Engineering and Knowledge Engineering (SEKE'04)*, pages 336–341. Knowledge Systems Institute, 2004.

[69] Tudor Gîrba, Stèphane Ducasse, and Michele Lanza. Yesterday's Weather: Guiding Early Reverse Engineering Efforts by Summarizing the Evolution of Changes. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 40–49, Washington, DC, USA, 2004. IEEE Computer Society.

[70] Tudor Gîrba and Stþhane Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(3):207–236, 2006.

[71] Neil M. Goldman. Smiley - an interactive tool for monitoring inter-module function calls. In *Proceedings of the 8th International Workshop on Program Comprehension (IWPC'00)*, pages 109–118. IEEE Computer Society, 2000.

[72] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a Call Graph Execution Profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[73] Dick Grune, Brian Berliner, Jeff Polk, Larry Jones, Derek R. Price, Mark D. Baushke, and et al. The Mozilla Organization. Concurrent Versions System, 1986–2006. *http://ftp.gnu.org/non-gnu/cvs/* [30 June 2006].

[74] Thomas Gschwind, Johann Oberleitner, and Martin Pinzger. Using Run-Time Data for Program Comprehension. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pages 245–250. IEEE Computer Society Press, May 2003.

[75] George Yanbing Guo, Joanne M. Atlee, and Rick Kazman. A Software Architecture Reconstruction Method. In *WICSA1: Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, pages 15–34, Deventer, The Netherlands, 1999. Kluwer, B.V.

[76] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. Compression Techniques to Simplify the Analysis of Large Execution Traces. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, pages 159–160. IEEE Computer Society, June 2002.

[77] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. A survey of Trace Exploration Tools and Techniques. In *CASCON '04: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 42–55. IBM Press, 2004.

[78] Richard C. Holt. Structural Manipulations of Software Architecture Using Tarski Relational Algebra. In *WCRE '98: Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, page 210, Washington, DC, USA, 1998. IEEE Computer Society.

[79] Richard C. Holt. Software architecture abstraction and aggregation as algebraic manipulations. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 5. IBM Press, 1999.

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

[81] Mehdi Jazayeri. On Architectural Stability and Evolution. In *Proceedings of the 7th Ada-Europe International Conference on Reliable Software Technologies*, pages 13–23, London, UK, 2002. Springer-Verlag.

[82] Mehdi Jazayeri, Alexander Ran, and Frank van der Linden. *Software Architecture for Product Families: Principles and Practice*. Addison Wesley, 2000.

[83] Dean F. Jerding and John T. Stasko. The Information Mural: A Technique for Displaying and Navigating Large Information Spaces. In *Proceedings of the 1995 IEEE Symposium on Information Visualization (INFOVIS'95)*, pages 43–50. IEEE Computer Society, October 1995.

[84] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[85] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study, CMU/SEI-90-TR-021. Technical report, Software Engineering Institute, Carnegie Mellon University, 1990. page 8.

[86] Rick Kazman and S. Jeromy Carrière. Playing Detective: Reconstructing Software Architecture from Available Evidence. *Automated Software Engineering*, 6(2):107–138, April 1999.

[87] Chris F. Kemerer and SandraSlaughter. An Empirical Approach to Studying Software Evolution. *IEEE Transactions on Software Engineering*, 25(4):493–509, July/August 1999.

[88] Gregor Kiczales and Erik Hilsdale. Aspect-Oriented Programming. In *ESEC/FSE-9: Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, page 313, New York, NY, USA, 2001. ACM Press.

[89] Miryung Kim and David Notkin. Using a Clone Genealogy Extractor for Understanding and Supporting Evolution of Code Clones. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[90] Sunghun Kim, Thomas Zimmermann, Miryung Kim, Ahmed Hassan, Audris Mockus, Tudor Gîrba, Martin Pinzger, E. James Whitehead, Jr., and Andreas Zeller. TA-RE: an Exchange Language for Mining Software Repositories. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 22–25, New York, NY, USA, 2006. ACM Press.

[91] Joseph. B. Kruskal and Myron Wish. Multidimensional Scaling. *Quantitative Applications in the Social Sciences*, 11, 1978.

[92] Michele Lanza. The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques. In *Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE'01)*, pages 37–42. ACM Press, September 2001.

[93] Michele Lanza and Stèphane Ducasse. Understanding Software Evolution using a Combination of Software Visualization and Software Metrics. In *LMO 2002 Proceedings (Languages et Modeles a Objets)*, pages 135–149. Hermes Publications, 2002.

[94] Michele Lanza and Stèphane Ducasse. Polymetric Views - A Lightweight Visual Approach to Reverse Engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, September 2003.

[95] Meir Manny Lehman. Laws of Software Evolution Revisited. In *Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124. Springer-Verlag, 1996.

[96] Todd A. Letsche and Michael W. Berry. Large-scale information retrieval with latent semantic indexing. *Information Sciences*, 100:105–137, August 1997.

[97] Christian Lindig. Compute Concept Lattice from Relation, 1998. *http://www.eecs.harvard.edu/~lindig/src/concepts.html* [30 June 2006].

[98] Christian Lindig and Gregor Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *ICSE'97: Proceedings of the 19th international conference on Software engineering*, pages 349–359, New York, NY, USA, 1997. ACM Press.

[99] Christian Lindig and Gregor Snelting. *Begriffliche Wissensverarbeitung. Methoden und Anwendungen*, chapter Formale Begriffsanalyse im Software Engineering. Springer, 1999.

[100] Chung-Horng Lung, Marzia Zaman, and Amit Nandi. Applications of clustering techniques to software partitioning, recovery and restructuring. *Journal of Systems and Software*, 73(2):227–244, October 2004.

[101] Jonathan I. Maletic and Michael L. Collard. Supporting Source Code Difference Analysis. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 210–219, Washington, DC, USA, 2004. IEEE Computer Society.

[102] Dane Marjanovic. Release History Meta Modeling. Master's thesis, Department of Informatics, University of Zurich, 2006.

[103] Nenad Medvidovic and Vladimir Jakobac. Using software evolution to focus architectural recovery. *Automated Software Engineering*, 13(2):225–256, 2006.

[104] Florian Mitter. Tracking Source Code Propagation in Software System via Release History Data and Code Clone Detection. Master's thesis, Technical University of Vienna, April 2006.

[105] Audris Mockus, Roy T. Fielding, and James 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.

[106] Leon Moonen. Generating Robust Parsers using Island Grammars. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 13, Washington, DC, USA, 2001. IEEE Computer Society.

[107] Hausi A. Müller and Karl Klashinsky. Rigi-A System for Programming-in-the-large. In *ICSE '88: Proceedings of the 10th International Conference on Software Engineering*, pages 80–86, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[108] Andreas Noack. An Energy Model for Visual Graph Clustering. In *Proceedings of the 11th International Symposium on Graph Drawing (GD 2003)*, volume 2912 of *Lecture Notes in Computer Science*, pages 425–436. Springer-Verlag, 2004.

[109] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992.

[110] David Lorge Parnas. Software aging. pages 279–287, May 1994.

[111] Martin Pinzger. *ArchView - Analyzing Evolutionary Aspects of Complex Software Systems*. PhD thesis, Vienna University of Technology, May 2005.

[112] Martin Pinzger, Michael Fischer, and Harald Gall. Towards an Integrated View on Architecture and its Evolution. In *Proceedings of the Software Evolution through Transformations: Model-based vs. Implementation-level Solutions (SETra'04)*, volume 127, pages 183–196. Elsevier Science Publishers: Utrecht, Netherlands, 2004.

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

[114] Martin Pinzger, Harald Gall, Michael Fischer, and Michele Lanza. Visualizing Multiple Evolution Metrics. In *Proceedings Symposium on Software Visualization (SoftVis'05)*, May 2005.

[115] Uta Priss. Formal Concept Analysis in Information Science. *Annual Review of Information Science and Technology*, 40:521–543, 2006. to appear.

[116] Elke Pulvermüller, Andreas Speck, James O. Coplien, Maja D'Hondt, and Wolfgang DeMeuter. Feature Interaction in Composed Systems. In *Proceedings Object-Oriented Technology ECOOP 2001 Workshop Reader*, volume 2323 of *Lecture Notes in Computer Science*, pages 86–97. Springer, June 2001.

[117] Jacek Ratzinger, Michael Fischer, and Harald Gall. EvoLens: Lens-View Visualizations of Evolution Data. In *Proceedings of the 8th International Workshop on Principles of Software Evolution (IWPSE'05)*. IEEE CS Press, September 2005.

[118] Jacek Ratzinger, Michael Fischer, and Harald Gall. Improving Evolvability through Refactoring. In *MSR 2005 Workshop Proceedings (MSR'05)*. ACM Digital Library, May 2005.

[119] Richard M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection (GCC)*. Free Software Foundation, 2004.

[120] Claudio Riva. *View-based Software Architecture Reconstruction*. PhD thesis, Vienna University of Technology, October 2004.

[121] Claudio Riva and Christian Del Rosso. Experiences with Software Product Family Evolution. In *Proceedings Sixth International Workshop on Principles of Software Evolution (IWPSE'03)*, pages 161–169. IEEE Computer Society Press, September 2003.

[122] Janice Singer, Robert Elves, and Margaret-Anne Storey. NavTracks: Supporting Navigation in Software Maintenance. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 325–334, Washington, DC, USA, 2005. IEEE Computer Society.

[123] Software Composition Group, University of Berne. The FAMIX 2.0 Specification, August 1999. *http://www.iam.unibe.ch/~famoos/FAMIX/* [30 June 2006].

[124] Software Engineering Group, Department of Informatics, University of Zurich. COSE - Controlling Software Evolution, 2005. *http://www.inf.unisi.ch/projects/cose/* [30 June 2006].

[125] Margaret-Anne D. Storey, K. Wong, F. D. Fracchia, and Hausi A. Müller. On Integrating Visualization Techniques for Effective Software Exploration. In *Proceedings of the 1997 IEEE Symposium on Information Visualization (InfoVis '97)*, pages 38–45. IEEE Computer Society, 1997.

[126] E. Burton Swanson. The dimensions of maintenance. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE)*, pages 492–497. IEEE Computer Society Press, October 1976.

[127] Nikita Synytskyy, James R. Cordy, and Thomas R. Dean. Robust Multilingual Parsing Using Island Grammars. In *CASCON '03: Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, pages 266–278. IBM Press, 2003.

[128] Ladan Tahvildari, Richard Gregory, and Kostas Kontogianni. An Approach for Measuring Software Evolution Using Source Code Features. In *Proceedings of the Sixth Asia Pacific Software Engineering Conference*, page 10. IEEE Computer Society, 1999.

[129] Alfred Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6:73–89, 1941.

[130] Christopher M. B. Taylor and Malcolm Munro. Revision Towers. In *Proceedings 1st International Workshop on Visualizing Software for Understanding and Analysis*, pages 43–50. IEEE Computer Society Press, June 2002.

[131] TechExcel Inc. Issue Tracking and Process Management. *http://www.techexcel.com/products/devtrack.html* [30 June 2006].

[132] Subrina Anjum Tonu, Azin Ashkan, and Ladan Tahvildari. Evaluating Architectural Stability Using a Metric-Based Approach. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 261–270, Washington, DC, USA, 2006. IEEE Computer Society.

[133] Frank van der Linden, editor. *Development and Evolution of Software Architectures for Product Families. Second International ESPRIT ARES Workshop, Las Palmas de Gran Canaria, Spain*, volume 1429 of *Lecture Notes in Computer Science*. Springer-Verlag Heidelberg, 1998.

[134] Frank van der Linden, editor. *Software Architectures for Product Families: International Workshop IW-SAPF-3, Las Palmas de Gran Canaria, Spain*, volume 1951 of *Lecture Notes in Computer Science*. Springer-Verlag Heidelberg, 2000.

[135] Frank van der Linden, editor. *Software Product Family Engineering: 4th International Workshop, PFE 2002, Bilbao, Spain*, volume 2290 of *Lecture Notes in Computer Science*. Springer-Verlag Heidelberg, 2002.

[136] Frank van der Linden, editor. *Software Product-Family Engineering: 5th International Workshop, PFE 2003, Siena, Italy*, volume 3014 of *Lecture Notes in Computer Science*. Springer-Verlag Heidelberg, 2004.

[137] Filip Van Rysselberghe and Serge Demeyer. Reconstruction of Successful Software Evolution Using Clone Detection. In *Proceedings Sixth International Workshop on Principles of Software Evolution (IWPSE'03)*, pages 126–130. IEEE Computer Society Press, September 2003.

[138] T. A. Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, page 33, Washington, DC, USA, 1997. IEEE Computer Society.

[139] Norman Wilde, Juan A. Gomez, Thomas Gust, and Douglas Strasburg. Locating User Functionality in Old Code. In *Proceedings International Conference on Software Maintenance*, pages 200–205. IEEE Computer Society Press, November 1992.

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

[141] Kenny Wong. *The Rigi User's Manual - Version 5.4.4*, 1998. *http://ftp.rigi.csc.uvic.ca/pub/rigi/doc/rigi-5.4.4-manual.pdf\refcheckdate*.

[142] Jingwei Wu, Claus W. Spitzer, Ahmed E. Hassan, and Richard C. Holt. Evolution Spectrographs: Visualizing Punctuated Change in Software Evolution. In *Proceedings 7th International Workshop on Principles of Software Evolution (IWPSE'04)*, pages 57–66. IEEE Computer Society Press, September 2004.

[143] Zhenchang Xing and Eleni Stroulia. Understanding Class Evolution in Object-Oriented Software. In *Proceedings 12th IEEE International Workshop on Program Comprehension*, pages 34–43. IEEE Computer Society, June 2004.

[144] Tetsuo Yamamoto, Makoto Matsushita, Toshihiro Kamiya, and Katsuro Inoue. Measuring Similarity of Large Software Systems Based on Source Code Correspondence. In *Proceedings of the 6th International Conference on Product Focused Software Process Improvement (PROFES'05)*, June 2005. to appear.

[145] Stephen S. Yau, J. S. Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *The IEEE Computer Society's Second International Computer Software and Applications Conference*, pages 60–65. IEEE Press, November 1978.

[146] Annie T. T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting Source Code Changes by Mining Change History. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[147] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1979.

[148] Thomas Zimmermann, Stephan Diehl, and Andreas Zeller. How History Justifies System Architecture (or not). In *Proceedings Sixth International Workshop on Principles of Software Evolution (IWPSE'03)*, pages 73–83. IEEE Computer Society Press, September 2003.

[149] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining Version Histories to Guide Software Changes. In *Proceedings 26th International Conference on Software Engineering (ICSE)*, pages 563–572. ACM Press, May 2004.