Available online at www.sciencedirect.com
SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 127 (2005) 183–196
www.elsevier.com/locate/entcs

Towards an Integrated View on Architecture and its Evolution

Martin Pinzger[1] Harald Gall[2]

Department of Informatics, University of Zurich

Michael Fischer[3]

Distributed Systems Group, Vienna University of Technology

Abstract

Information about the evolution of a software architecture can be found in the source basis of a project and in the release history data such as modification and problem reports. Existing approaches deal with these two data sources separately and do not exploit the integration of their analyses. In this paper, we present an architecture analysis approach that provides an integration of both kinds of evolution data. The analysis applies fact extraction 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 architectural views using binary relational algebra. These integrated architectural views show intended and unintended couplings between architectural elements, hence pointing software engineers to locations in the system that may be critical for on-going and future maintenance activities. We demonstrate our analysis approach using a large open source software system.

Keywords: software evolution analysis, software architecture, architectural views

1 Introduction

Higher-level views on the architecture of software systems aid engineers in evolving and maintaining software systems. Typically, these views are de-

[1] Email:pinzger@ifi.unizh.ch
[2] Email:gall@ifi.unizh.ch
[3] Email:M.Fischer@infosys.tuwien.ac.at

doi:10.1016/j.entcs.2004.08.043

picted as graphs whereas nodes represent the architectural elements and edges the relationships between them. In particular, relationships represent dependencies between architectural elements that lead to *coupling* between these elements. In theory, strongly coupled architectural elements are more likely to be modified together than are loosely coupled elements. Therefore, architectural designs concentrate on encapsulating common behavior within an architectural element, consequently increasing cohesion and lower coupling [2].

An *architectural element* in the context of our work is a *software module* that results from the decomposition of a software system into implementation units. According to Clements et al. [6] 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 [2].

However, in practice evolution often draws a different picture revealing couplings between architectural elements that were not intended by the architectural design. Reasons for this are manifold: shortcomings in the initial design; the architecture has not been implemented in the way it was designed; or architecture drift due to frequent modifications to the implementation. In fact, these dependencies—for example constant changes crossing module boundaries—hinder the effective maintenance and evolution of software systems. Therefore, locating them in the current implementation to facilitate the application of directed refactorings to resolve these couplings would be beneficial.

In this paper we focus on analyzing dependencies between architectural elements and introduce the architecture analysis approach *ArchEvo*. ArchEvo enriches source code models extracted from source code and execution traces with logical coupling data obtained from configuration management systems [11,12]. We refer to logical coupling as: *Two source code entities (e.g. files) are logically coupled if a modification to the implementation affected both source code entities over a significant number of releases.*

The data sources are integrated into a common directed attributed graph from which ArchEvo abstracts higher-level views using architecture recovery [15]. 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 architecture and its evolution that points software architects and engineers to shortcomings in the design and implementation that should undergo directed refactorings.

The remainder of this paper is organized as follows: In Section 2 we introduce the architecture analysis approach ArchEvo and describe the building of the integrated fact repository and the abstraction of higher-level views. Section 3 describes our findings concerning the coupling relationships with a selected set of software modules and features of the open source web browser Mozilla. In Section 4 we present related work and Section 5 draws some conclusions and indicates future work.

2 ArchEvo Approach

Analyzing the dependencies between architectural elements is a key issue when analyzing the architecture of software systems. Recent research in analyzing these dependencies (i.e. coupling) 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 [3]. In our recent research we concentrated on investigating configuration management data including version, change, and defect data to obtain information about logical couplings (i.e. hidden dependencies) between source code units [10,12].

The ArchEvo approach presented in this paper is a combination of both approaches mentioned before and extends them by analyzing coupling relationships on the architectural level. Figure 1 depicts the process followed by ArchEvo. The process steps are described in the following subsections.

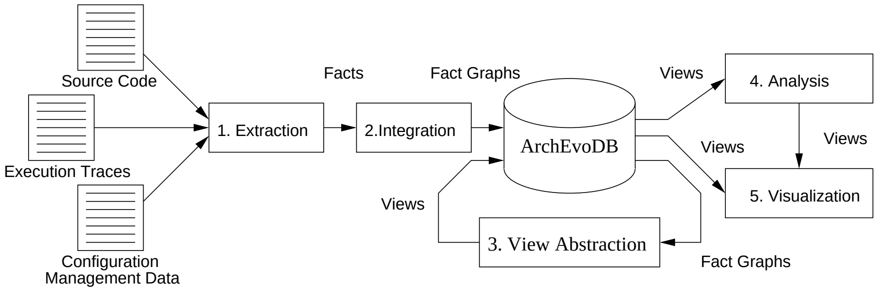

Fig. 1. ArchEvo architecture evolution analysis process.

2.1 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.

Release history data of a software project is obtained from configuration management systems in the form of modification reports. They are generated by versioning systems such as cvs [5] or Subversion [7] and deliver data about changes made to source files. These reports are parsed for relevant data used to identify logical coupling relationships. Following our definition of logical coupling we establish a logical coupling relationship between two entities if there is a modification report that references both entities [11]. Currently, logical coupling is detected on the level of source files which is sufficient for our current 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 1). ArchEvo uses the Rigi Standard Format (RSF) 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).

Facts obtained from configuration management systems are stored to a relational database. 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 configuration management data stored in the relational database is converted to a fact graph. Nodes of this graph represent source files and modification reports and edges represent established couples relationships 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.

2.2 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 [1] 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 and analysis tasks.

2.3 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 ArchEvo is based on the approach presented by Holt et al. in [13], 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 [9].

Relationships between architectural elements and abstraction measures are computed using binary relational algebra. Currently, we use the grok tool [8] 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 1 defines the ArchEvo abstraction algorithm that is applied to the directed attributed fact graph.

Algorithm 1 *ArchEvo abstraction algorithm*

```
1: foreach entity pair (A,B) do
2:     setA := entities contained by A
3:     setB := entities contained by B
4:     relsAB := relationships of type T between setA and setB
5:     if #relsAB >0 then
6:         rel := create relationship of type T between A and B
7:         measures := compute abstraction measures of rel
8:     end if
9: end foreach
```

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 1 lists the measures computed for abstracted relationships (these measures also apply to other levels of abstraction).

Table 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

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.

2.4 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.

In the next section we describe our analysis of the open source web browser Mozilla [18].

3 ArchEvo Views

The outcome of the ArchEvo architecture analysis process are views that can be used by the user to identify starting points for minor changes on lower level or major re-design phases. To demonstrate our ArchEvo approach we applied it to the open source web browser Mozilla version 1.3a (released December 2002). At this time the Mozilla application suite comprised more than 10.400 source files in C/C++ containing about 3.700.000 source text lines distributed over 2.500 directories and more than 90 software modules. Starting from Mozilla's design documentation we focused our analysis 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 2 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.

Table 2. Selected Mozilla modules and their source code directories

Module	Source Directories
MathML	layout/mathml
New Layout Engine	layout/base, layout/build, layout/html
XPToolkit	content/xul, layout/xul
Document Object Model (DOM)	content/base, content/events, content/html/content, content/html/document, dom
New HTML Style System	content/html/style, content/shared
XML	content/xml, expat, extensions/xmlextras
XSLT	content/xsl, extensions/transformiix

Subgoals in our analysis 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 1

Based on these views and measurements we analyzed the as-implemented architecture 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 [3].

The following sections report on our findings about the selected modules listed in Table 2.

3.1 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 2 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 (i.e., pairwise changes) between these modules and indicated as straight, cyan/solid line. Since all modules are coupled with each other through "administrative noise", weaker couplings are omitted. As threshold we used 10% of the maxium coupling (`DOM – New Layout Engine`) which in turn comprises more than 48.000 pairwise modifications. The "administrative noise" mentioned above typically involves several hundred files. Messages left in the description field of these administrative modifications are for instance "license foo" (with 7.961 referenced files), "printfs and console window info needs to be boiled away for release builds" (1.135 files), or "Clean up SDK includes" (888 files).

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 two small graphs in Figure 2 on the right side show the same coupling graph but the inherits and aggregates relationships. 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. Directed refactorings can be applied to these locations to improve the design and reduce the pairwise modifications in the future. However, module views are abstract rep-

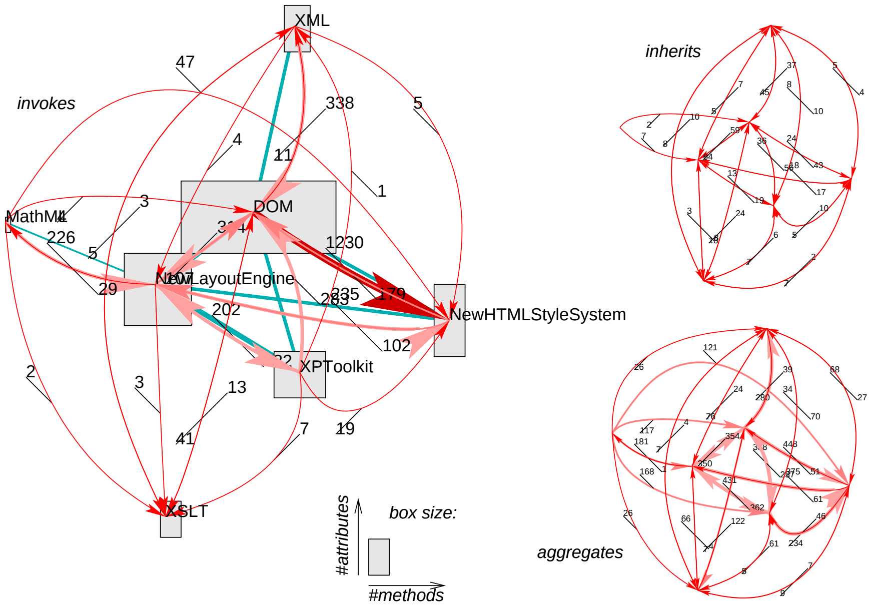

Fig. 2. Invokes, inherits and aggregates between selected modules

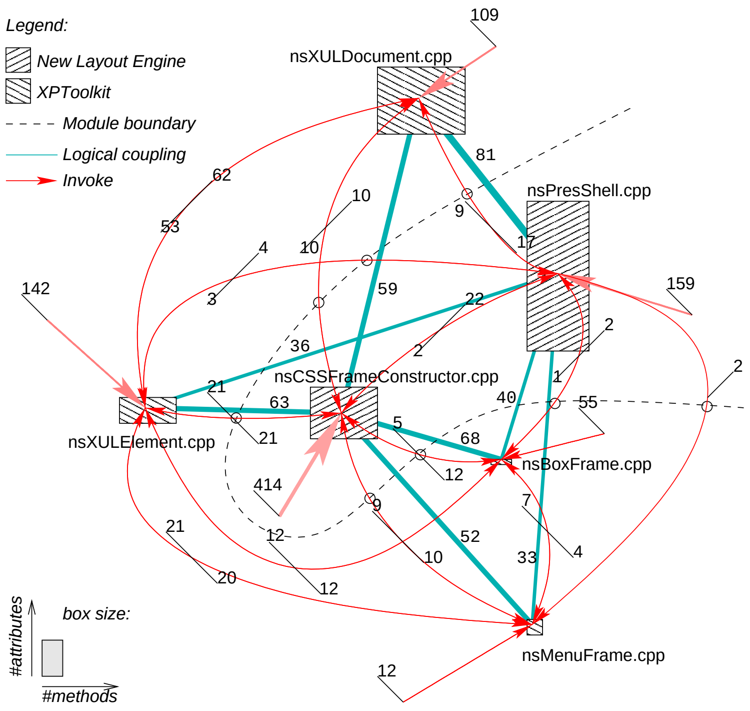

Fig. 3. Invocations between files (Modules `XPToolkit` and `New Layout Engine`)

resentations of underlying source code data. Therefore, deeper insights into the coupling dependencies are mandatory to refactor them.

3.2 A 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 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 4 calls from *nsXULElement.cpp* to *nsPresShell.cpp* and 3 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. Knowing the critical source code entities the user can direct his refactoring activities to these entities to improve the *as-implemented design* of the system under study.

4 Related Work

Related work ranges from evolution analysis to architecture recovery and coupling analysis approaches. Concerning evolution analysis Zimmermann et al. inspected release history data of several software systems for logical coupling between source code entities [20]. They drew the conclusion that augmentation of architectural data with evolutionary information could reveal new otherwise hidden dependencies between source code entities. Even though a number of other work used release history data as well, a detailed evaluation of the correlation between source model entities and the properties of logical coupling is still missing.

Hsi and Potts [14] studied the evolution of user-level structures and operations of a large commercial text processing software package over three releases. Based on user interface observations they derived three primary views describing the user interface elements (morphological view), the operations a user can call (functional view), and the static relationships between objects in the problem domain (object view). As this approach does not consider a thorough code analysis, user interface issues are usually not taken into account during code analysis, a fusion with methods regarding code and release history data would yield good results in feature evolution analysis.

In [12] we examined the structure of a *Telecommunications Switching Software* (TSS) over more than 20 releases to identify logical coupling between system and subsystems. As step towards combination of abstract concepts such as feature information and logical coupling we investigated in [10] the reflection of qualified release history data onto different source code model entities. Release history data comprised modification report plus problem tracking data. The source code model data representing different features were derived from source code using runtime information. The verification of properties of the underlying source code model such as aggregation etc. were beyond the scope of this work.

In [3] Briand et al. discussed a unified framework for coupling measurement in object-oriented systems based on source model entities. Based on this metrics they verified in [4] the coupling measurements on file level using statistical methods and logical coupling information based on "ripple effects" [19]. A classification of logical coupling information to verify the properties of coupling measurement has been omitted. In our approach we go further and use file level information to abstract onto higher architectural views such as module view as well.

Wilkie and Kitchenham investigated the correlation of coupling between objects (CBO) and change ripples of a C++ application [16,17]. Their work was focused on class level properties in contrast to our work which is primarily focused on higher, more abstract architectural views.

Concerning architecture recovery several related approaches exists such as, for example, described by Holt et al. in [13] and Feijs et al. in [9]. Both approaches deal with abstracting lower-level source code information. The abstraction algorithm used by ArchEvo is based on them. However, in extension to these approaches ArchEvo takes into account modification and problem report data. Further, ArchEvo concentrates on the computation of measures for abstracted elements and relationships.

5 Conclusions and Future Work

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 extraction 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 architectural views using binary relational algebra. These integrated architectural views show intended and unintended couplings between architectural elements, hence pointing software architects to locations in the system that may be critical for on-going and future maintenance activities. Thus, ArchEvo's contribution is the abstraction of detailed source code model data and evolutionary information onto more abstract levels. This supports the reflection of the concrete implementation at its design level with focus on the coupling dependencies between architectural elements. Details of selected coupling dependencies are obtained by decreasing the level of abstraction. Consequently, the benefits of ArchEvo are in (1) graphically highlighting locations of design erosion in the as-implemented architecture that led to logical couplings; and (2) revealing the implementation details potentially causing them.

Further benefits of the ArchEvo approach are: (a) compact graphical representation of architectural data together with evolutionary information; (b) location of areas with frequent modifications; (c) providing good views onto dependencies between different elements of the source code model; (d) support for different views on arbitrary abstraction levels such as file-, component-, or module-level; and (e) quick identification of tightly coupled files or modules through their placement in the graphical representation. Finally, our approach has been validated using the large open source software project of Mozilla.

Interesting areas for future work are qualitative and quantitative analysis of the properties of logical and architectural coupling such as inter- and intra-module coupling, evaluation of properties of transitive dependencies, i.e., logical coupling but no direct architectural dependency between different source code model entities, extending this methodology to support change impact analysis of large scale software, integration of source code model deltas between different releases to automatically classify the type of modification such as interface changes, add/remove invocation relationship, aggregation etc. (statement level analysis). Another perspective is the integration of problem report data into the analysis process to deliver further hints for the search of error prone entities within the abstracted views.

Acknowledgments

We thank the Mozilla developers for providing all their data for this case study. The work described in this paper was supported by the Austrian Ministry for Infrastructure, Innovation and Technology (BMVIT), The Austrian Industrial Research Promotion Fund (FFF), and the European Commission in terms of the EUREKA 2023/ITEA projects CAFÉ and FAMILIES (http://www.infosys.tuwien.ac.at/Cafe/).

References

[1] "The FAMIX 2.0 Specification," 2.0 edition (1999), http://www.iam.unibe.ch/~scg/Archive/famoos/FAMIX/.

[2] Bass, L., P. Clements and R. Kazman, "Software Architecture in Practice," Addison-Wesley, 2003, 2nd edition.

[3] Briand, L. C., J. W. Daly and J. K. Wüst, *A unified framework for coupling measurement in object-oriented systems*, Journal of IEEE Transactions on Software Engineering **25** (1999), pp. 91–121.

[4] Briand, L. C., J. Wüst and H. Lounis, *Using coupling measurement for impact analysis in object-oriented systems*, in: *Proceedings of the IEEE International Conference on Software Maintenance* (1999), pp. 475–482.

[5] Cederqvist, P., "Version Management with CVS," (1992), http://www.cvshome.org/docs/manual.

[6] Clements, P., F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord and J. Stafford, editors, "Documenting Software Architectures: Views and Beyond," Addison-Wesley Professional, 2002.

[7] Collins-Sussman, B., B. W. Fitzpatrick and C. M. Pilato, "Version Control with Subversion," (2004), http://svnbook.red-bean.com/svnbook-1.1.

[8] Fahmy, H. and R. C. Holt, *Software architecture transformations*, in: *Proceedings of the IEEE International Conference on Software Maintenance* (2000), pp. 88–96.

[9] Feijs, L., R. Krikhaar and R. van Ommering, *A relational approach to support software architecture analysis*, Journal of Software Practice and Experience **28** (1998), pp. 371–400.

[10] Fischer, M. and H. Gall, *Visualizing feature evolution of large-scale software based on problem and modification report data*, Journal of Software Maintenance and Evolution **16** (2004).

[11] Fischer, M., M. Pinzger and H. Gall, *Populating a release history database from version control and bug tracking systems*, in: *Proceedings of the IEEE International Conference on Software Maintenance* (2003), pp. 23–32.

[12] Gall, H., K. Hajek and M. Jazayeri, *Detection of logical coupling based on product release history*, in: *Proceedings of the IEEE International Conference on Software Maintenance* (1998), pp. 190–198.

[13] Holt, R. C., *Structural manipulations of software architecture using tarski relational algebra*, in: *Proceedings of the IEEE Working Conference on Reverse Engineering* (1998), pp. 210–219.

[14] Hsi, I. and C. Potts, *Studying the evolution and enhancement of software features*, in: *Proceedings of the 2000 IEEE International Conference on Software Maintenance*, 2000, pp. 143–151.

[15] Pinzger, M., M. Fischer, M. Jazayeri and H. Gall, *Abstracting module views from source code*, in: *Proceedings of the IEEE International Conference on Software Maintenance* (2004).

[16] Wilkie, F. G. and B. A. Kitchenham, *Coupling measures and change ripples in c++ application software*, Journal of Systems and Software **52** (2000), pp. 157–164.

[17] Wilkie, F. G. and B. A. Kitchenham, *An investigation of coupling, reuse and maintenance in a commercial c++ application*, Journal of Information and Software Technology **43** (2001), pp. 801–812.

[18] *Mozilla open-source web browser*, http://www.mozilla.org.

[19] Yau, S. S., J. S. Collofello and T. MacGregor, *Ripple effect analysis of software maintenance*, in: *Proceedings of the IEEE International Computer Software and Applications Conference* (1978), pp. 60–65.

[20] Zimmermann, T., S. Diehl and A. Zeller, *How history justifies system architecture (or not)*, in: *Proceedings of the IEEE 6th International Workshop on Principles of Software Evolution* (2003), pp. 73–83.