The Evolution Radar: Visualizing Integrated Logical Coupling Information

Marco D'Ambros, Michele Lanza, Mircea Lungu
Faculty of Informatics
University of Lugano, Switzerland
{marco.dambros, michele.lanza, mircea.lungu}@lu.unisi.ch

ABSTRACT

In software evolution research logical coupling has extensively been used to recover the hidden dependencies between source code artifacts. They would otherwise go lost because of the file-based nature of current versioning systems. Previous research has dealt with low-level couplings between files, leading to an explosion of data to be analyzed, or has abstracted the logical couplings to module level, leading to a loss of detailed information. In this paper we propose a visualization-based approach which integrates both file-level and module-level logical coupling information. This not only facilitates an in-depth analysis of the logical couplings at all granularity levels, it also leads to a precise characterization of the system modules in terms of their logical coupling dependencies.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Maintenance, Version Control, Reengineering, Reverse Engineering

General Terms

Measurements, Design

Keywords

Evolution, Logical Coupling, Visualization

1. INTRODUCTION

Versioning systems allow developers to record the history of a software project. The facilities given by versioning systems and the amount of data retrieved fostered the research field of software evolution [13], whose goal is to analyze the history of a software system and infer causes of its current problems, and possibly predict its future.

The history of a software system also holds information about the logical couplings. These are implicit and evolutionary dependency relationships between the artifacts of a system which, although potentially not structurally related, evolve together and are therefore linked to each other from a development process point of view. In short, logically coupled entities have changed together in the past and are thus likely to change in the future. Logical coupling information can therefore be used to predict the evolution of a software system. Moreover, logical coupling information reveals potentially misplaced artifacts in a software system, because entities that evolve together should be placed close to each other for cognitive reasons: A developer who modifies a file in a system could forget to modify related files because they are placed in other subsystems or packages.

In this paper we propose a technique to inspect logical coupling relationships, which integrates information both at a module-level (which subsystems are coupled with each other) and at a file-level (which files are responsible for the logical couplings). Our technique is based on a specific visualization that we named *Evolution Radar*. Visualization techniques have already been successfully used to study the evolution of software systems [1, 5, 10, 11, 14, 16, 17].

With our approach we tackle the following problems:

- How to present very large amounts of evolutionary information in an effective way.
- How to render logical coupling relationships in an intuitive way.
- How to enable a developer to study and inspect these relationships and to guide him to the files that are responsible for the logical couplings.

All the results and the examples presented in the Paper have been obtained by applying the presented visualization technique on the Mozilla (www.mozilla.org) case study.

Structure of the paper. In Section 2 we discuss the research that has been performed on logical coupling. In Section 3 we introduce our approach based on the *Evolution Radar* to render logical coupling information. We validate our technique on a large software system in Section 4 and we look at related work in Section 5. In Section 6 we conclude the Paper by summarizing our contributions and give an outlook on our future work in this field.

2. LOGICAL COUPLING

Logical coupling represents the implicit dependency relationship between two or more software artifacts that have been observed to frequently change together during the evolution of the system. This co-change information can either be present in the versioning system, or must be inferred by analysis. For example subversion marks co-changing files at commit time as belonging to the same *change set* while the files which are logically coupled must be inferred from the modification time of each individual file.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MSR'06, May 22–23, 2006, Shanghai, China.
Copyright 2006 ACM 1-59593-085-X/06/0005 ...$5.00.

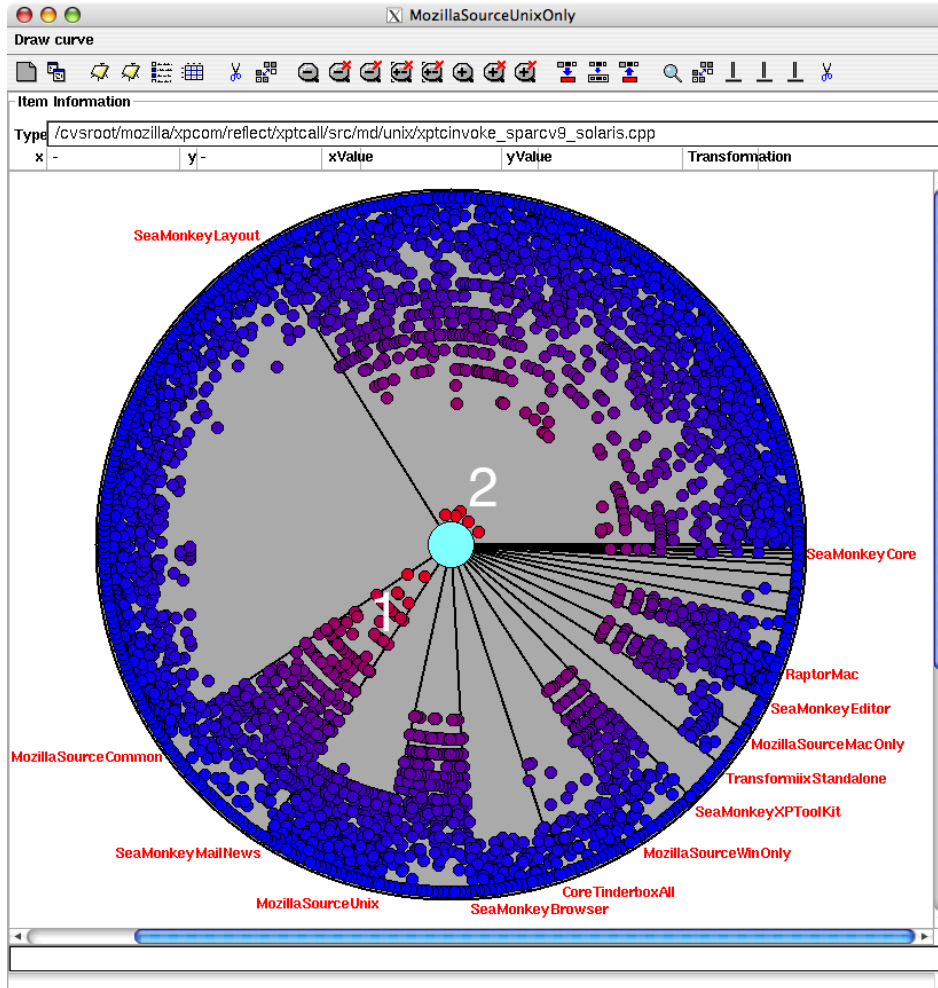


Figure 1: A sample Evolution Radar visualization of the core Mozilla modules.

The concept of logical coupling was first introduced by Gall *et al.* [7] to detect implicit relationships between modules. The technique that they proposed uses information from the CVS versioning control system to detect dependencies between the modules of a system. They used logical coupling to analyze the dependencies between the different modules of a large telecommunications software system and show that the approach can be used to derive useful insights on the architecture of the system. There are two reasons why the technique proved to be useful:

- It is more lightweight than structural analysis, as it needs to analyze a smaller amount of data, *i.e.*, only the data provided by the CVS log files. Moreover, as it works at text level, it can analyze systems written in multiple languages without the trouble of parsing and analyzing the data.
- It can reveal dependencies that are not structural, and therefore are not present in the code or in the documentation. These dependencies are the most troublesome and are prone to represent sources of bugs in software projects.

Later the same authors revisited the technique to work at a lower abstraction level. They detected logical couplings at class level [8] and validated it on 28 releases of an industrial software system. The authors showed through a case study that architectural weaknesses such as poorly designed interfaces and inheritance hierarchies could be detected based on logical coupling information.

Ratzinger *et al.* [15] used the same technique for analyzing the logical coupling at the class level with the aim of learning about, and improving the quality of the system. To accomplish this, they defined *code smells* based on the logical coupling between classes of the system.

Working at a finer granularity level, Zimmermann *et al.* [20] used the information about changes that are occurring together to predict entities that are likely to be modified when one is being modified.

The main problem with the mentioned approaches is that they either work at the architecture level, *i.e.*, without knowing which finer-grained entities cause the logical coupling, or they work at the file (or even finer) granularity level, *i.e.*, losing the global view of the system.

In this paper we propose an approach to overcome this shortcoming by means of a visualization technique called *Evolution Radar*, presented next.

3. THE EVOLUTION RADAR

The Evolution Radar is a visualization technique to render file-level and module-level logical coupling information in an integrated and interactive way, thus allowing the viewer to navigate and query the visualized data. It is implemented in BugCrawler [6].

3.1 Principles

The Evolution Radar (see Figure 1) visualizes the logical coupling of one module with the others. The module in focus is placed in the middle of a pie chart, where each sector represents one of the other modules. The size of each sector depicts the size of each module in terms of number of files. The modules are sorted according to this size metric.

The files of each of those modules are represented as colored circles and placed according to the logical coupling they have with the module placed in the center (the closer the files are to the center, the more coupled they are). The logical coupling can also be mapped on the color of the figures using a temperature representation: the hotter (from blue to red) the color is, the higher the value of the logical coupling measure is.

In the Evolution Radar it is possible to use different time interval combinations for the computation of the logical coupling and any combination of the position-color mapping. For example we can use the last year as time interval and map the resulting measure on the position, while for the color we compute the logical coupling considering the last month of the history of files.

3.2 Example

In the example Evolution Radar depicted in Figure 1 we see that one module (MozillaSourceCommon marked as 1) in the lower half of the radar has many files which are strongly coupled with the files in the center module (MozillaSourceUnixOnly). As a result we can say that these modules are strongly logically coupled.

Moreover, we see that the largest module (SeaMonkeyCore marked as 2) contains many files (the sector is big), but only a few are logically coupled with the center module. These files should be investigated to see whether they should be moved to the center module.

3.3 Logical Coupling Measure

We define the logical coupling between a file f and a module M as the maximum logical coupling between f and all the files belonging to M. The coupling between two files is defined as the number of "shared commits" they have, *i.e.,* the commits performed within a fixed time window[1]. This approach can be improved using a sliding time window as proposed by Zimmermann *et al.* in [19]. Using a sliding time window the obtained set of shared commits is a superset of the one obtained using a fixed time window. Thus the results found with our approach are still valid with the sliding time window, but with the latter it might be possible to find more of them.

To avoid outliers (files with a very high value of logical coupling with respect to the average) to deform our visualization, *i.e.,* pushing all the other figures to the boundary, we use a percentage value. We divide the number of shared commits by the average of the total number of commits of the two files. However, the percentage measure does not weigh the logical coupling with the absolute number of commits, implying that a file with 5 commits has the same value of a file with 100 commits if they have the same number of shared commits. A solution to this problem consists in multiplying the logical coupling percentage value with the logarithm of the total number of commits. Experiments with both the measures show that even with the log scale some entities are displaced too much from the center because of the outliers. Thus we choose the percentage value as logical coupling measure, solving the problem using a simple query engine. It allows the user to select and/or remove from the view all files having a number of commits below a given value.

[1]We use a 200 seconds time window, as used in [19].

3.4 Advantages

The Evolution Radar is interactive, *i.e.,* the user can zoom in on details, can select, inspect, remove single files, *etc.* to verify hypotheses such as whether certain files should be moved from one module to the other.

The Evolution Radar has several advantages regarding its visual expressiveness: It is rotation invariant like Chuah's time wheel visualizations [4]. It occupies a settable amount of screen space, *i.e.,* it is always possible to visualize the whole radar on screen, independent of its resolution. It does not visualize the coupling relationships as edges and therefore does not suffer from overplotting: The radar always remains intelligible, *i.e.,* it is easy to make out the heavily coupled modules which are displayed as "spikes" pointing to the center. It is also easy to make out single files responsible for the coupling which are placed close to the center. The Evolution Radar is applicable not only to modules, but also to any set of files.

4. MOZILLA EXPERIMENTS

The Evolution Radar helps in answering questions about the evolution of a system which are useful to developers, analysts, and project managers:

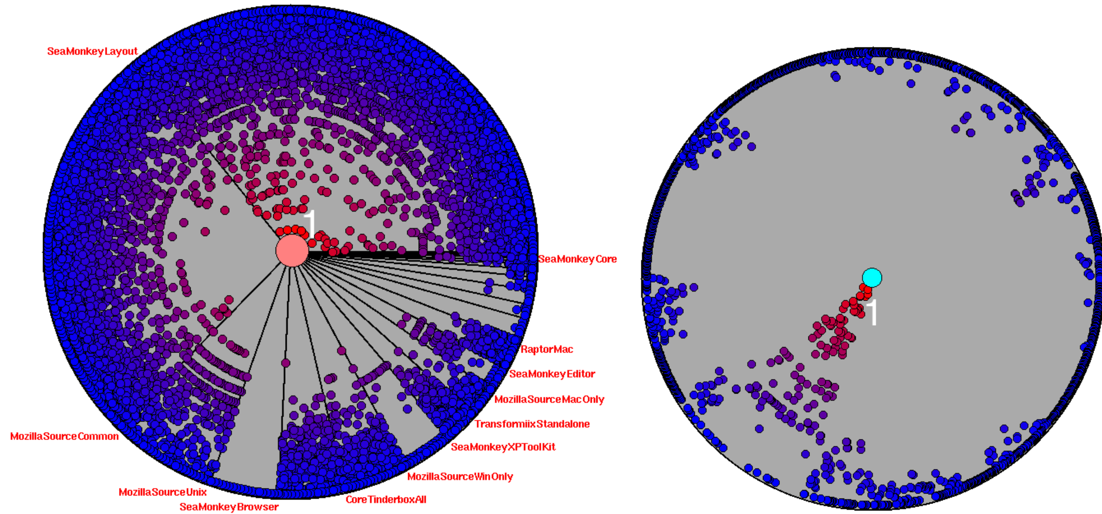
- *Developers* can use the technique to answer the question: "If I change this file, what others will I have to modify?". The Evolution Radar offers a visual way to assess the files that might change in the future based on the prediction offered by logical coupling. Due to the fine-grained level of the visualization, files can be inspected individually.

- *Analysts and project managers* can use the Evolution Radar to (i) understand the overall structure of the system in terms of module dependencies, (ii) examine the structure of these dependencies at the file granularity level and (iii) get an insight of the impact of changes on a module over other modules. This knowledge will help them in (i) localizing where refactorings should be applied, (ii) deciding whether certain files should be moved to other modules, and (iii) understanding the evolution of the logical coupling among modules.

In the remainder of this section we provide example scenarios of applying the Evolution Radar technique on 30'000 source code files in the Mozilla case study. For each example we mention which was the goal of the analysis and the potential stakeholders. Throughout the examples the color metric is the same as the distance metric unless otherwise specified.

4.1 Understanding SeaMonkeyMailNews

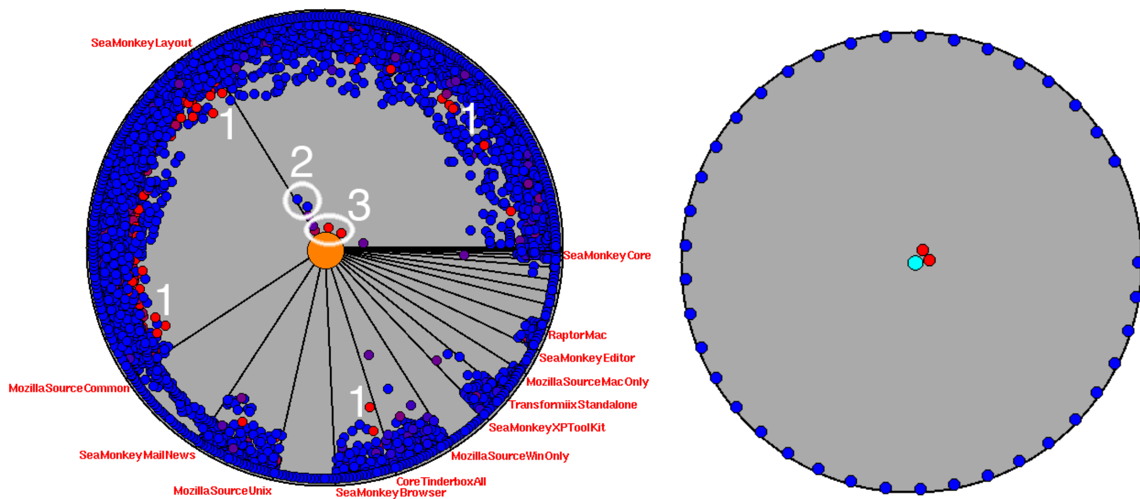
Target Audience: Analysts, project managers.
Goals. (1) To understand the dependencies between a module and all the other modules, (2) to understand the causes of these dependencies, and (3) to get an insight on the impact of changes regarding the target module.
Analysis. SeaMonkeyMailNews is a large module (1302 files) with strong dependencies with all the other modules in the system, especially with the largest module SeaMonkeyCore (7834 files). Figure 2(a) shows the Evolution Radar of SeaMonkeyMailNews. Many files are involved in the logical coupling between SeaMonkeyCore and SeaMonkeyMailNews. This information is useful but



(a) The Evolution Radar of the SeaMonkeyMailNews module.

(b) The details of the logical coupling between SeaMonkeyMailNews and SeaMonkeyCore.

Figure 2: Evolution Radars for SeaMonkeyMailNews.



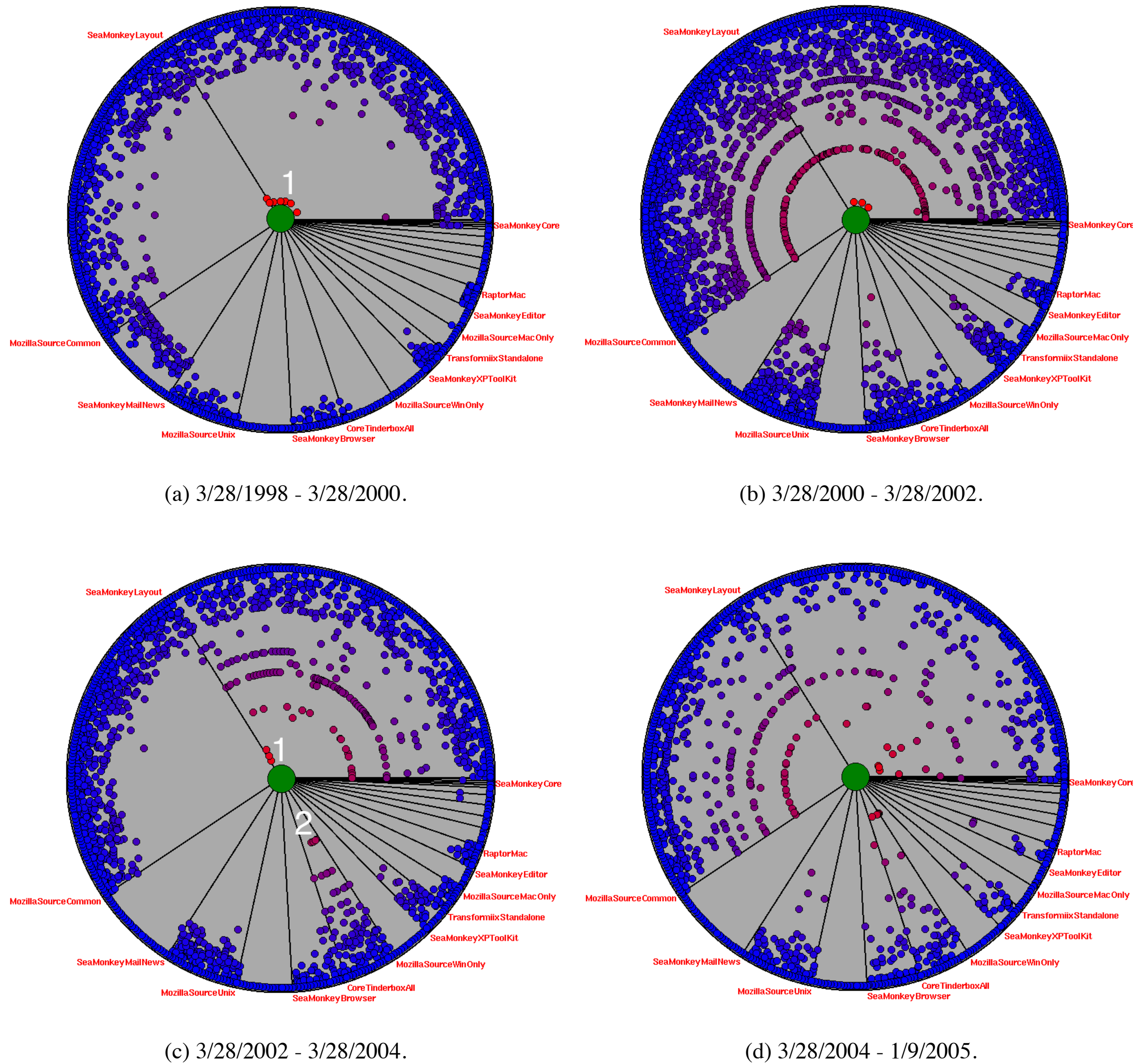
(a) The Evolution Radar of the ThunderbirdTinderbox module.

(b) The details of the logical coupling between ThunderbirdTinderbox and the `rdf/chrome/src/*` files.

Figure 3: Evolution Radars for ThunderbirdTinderbox.

still too coarse-grained. Thus we need to understand how the logical coupling is structured in terms of the individual files.

We refine the view of the logical coupling between modules by selecting the files closest to the center that are marked as (1) in Figure 2(a), and reapply the Evolution Radar for them. Now the group of selected files belonging to SeaMonkeyCore plays the role of the module in the center (represented as the cyan disc in Figure 2(b)), and the contents of SeaMonkeyMailNews, which was the previous center module, are scattered around them. As we can see from Figure 2(b) the logical coupling is due to the files marked as 1. All these files belong to the `mailnews/db/mork` directories tree, while the ones marked as 1 in Figure 2(a) belong to `db/mork`. These two hierarchies should be further inspected and, in case, merged and moved to the appropriate module.



(a) 3/28/1998 - 3/28/2000.

(b) 3/28/2000 - 3/28/2002.

(c) 3/28/2002 - 3/28/2004.

(d) 3/28/2004 - 1/9/2005.

Figure 4: The Evolution of the PhoenixTinderbox logical couplings.

4.2 ThunderbirdTinderbox Impact Analysis

Target Audience. Developers.
Goals. (1) To understand the change impact at the file level (*i.e.*, answer the question: "If a file changes in this module, what other files might have to change?").
Analysis. ThunderbirdTinderbox is a small module (42 files) but it has dependencies with most of the other Mozilla modules. As developers, we want the best candidates, *i.e.*, the files which have not only the strongest logical couplings but also the most recent ones. We tackle the problem by creating an Evolution Radar view which presents two types of logical coupling: (i) Coupling computed for the whole history of the system and (ii) coupling computed for the last 6 months only.

To present both types of information in the same figure we map the logical coupling based on the history of the last 6 months on the color of the discs representing files, while keeping the distance to represent the logical coupling for the whole history. In Figure 3(a) we can see three types of files:

- The files in group (3) are the most interesting, as they were always, and especially during the last 6 months, changed together with some files of the ThunderbirdTinderbox module.
- For the files marked with (1), the logical coupling is weak when computed for the whole history, but is strong when computed considering only recent changes.
- For the files in group (2), the inverse holds, because the coupling recently decreased (strong for the whole history, weak for the last six months).

To continue the analysis, we focus our attention on the files in group (3) only (all of which belong to the `rdf/chrome/src/` directory), because we want our candidate set to be small. We build another Evolution Radar (Figure 3(b)) using the set of files

in group (3) as the reference point and with the files belonging to the ThunderbirdTinderbox module scattered around it.

We find out that the logical coupling is due to two files only (`chrome/src/nsChromeURL.cpp` and `.h`) in the ThunderbirdTinderbox module. This means that if we want to modify the `nsChromeURL` files, it is very likely that we have to modify the files belonging to group 3 as well, while for all the other ThunderbirdTinderbox files we don't have this problem.

4.3 The Evolution of PhoenixTinderbox

Target Audience. Project managers, analysts.

Goals. (1) To obtain a high-level insight about the past evolution of the logical coupling relationships of a certain module during development phases, and (2) to understand whether the logical coupling relationship of a module is "ameliorating" or "degrading". It is degrading if the module is more and more logically coupled to the others, leading to maintenance problems and suggesting refactoring.

Analysis. The evolution of the logical coupling for the module PhoenixTinderbox (depicted in Figure 4) shows that the module went through diverse phases.

1. In the first phase from 1998 to 2000 it was decoupled from most of the system modules except SeaMonkeyCore because of the few files marked as (1).
2. Between 2000 and 2002 its architecture degraded since it became more coupled with other modules, namely: SeaMonkeyLayout, SeaMonkeyMailNews, SeaMonkeyBrowser, SeaMonkeyXPToolKit.
3. Between 2002 and 2004, possibly due to a restructuring phase, most of the logical couplings were reduced but the co-dependency with SeaMonkeyCore remained (marked as 1) and the one with CoreTinderboxAll increased (marked as 2).
4. In the last phase the architecture degraded again since (i) the dependencies with SeaMonkeyCore were reduced but still remained, (ii) the logical coupling with SeaMonkeyLayout became strong again and (iii) the dependencies with SeaMonkeyMailNews, CoreTinderboxAll and SeaMonkeyEditor were slightly increased.

5. RELATED WORK

Since Section 2 already introduced related work on logical coupling, this section presents work related to software evolution visualization.

A similar approach to visualize logical coupling has been presented by Pinzger et al. [14] with Kiviat Diagrams. As a difference they do not visualize file-level information but use surfaces to depict complete releases, while in our visualization we depict all evolving files in one diagram. Another difference is that they represent the coupling as edges between the visible modules.

The graph based representation in which entities involved in logical coupling were nodes in a graph and coupling was represented as edges between them was used since the first publications related to logical coupling [7, 8]. However, the problem with this representation is that it either represents only modules, and then it is too coarse grained, or it represents modules and files, but then it does not scale to large systems.

A visual data-mining tool to represent both binary association rules and n-ary association rules is EPOsee [3]. The tool adapts standard visualization techniques for association rules to also display hierarchical information.

Chuah and Eick present a way to visualize project information through glyphs called infobugs. Glyphs are graphical objects representing data through visual parameters. Their infobug glyph's parts represent data about software [4]. The difference with respect to our work is that they use glyphs to view project management data, while our work focuses on describing how a module is logically coupled to the others. One common advantage is that both approaches are rotation invariant.

Lanza's Evolution Matrix [12] visualizes the system's history in a matrix in which each row is the history of a class. A cell in the Evolution Matrix represents a class and the dimensions of the cell are given by evolutionary measurements computed on subsequent versions. The evolution matrix does not represent any relationship between the evolving entities.

Bayer [2] computes a co-change graph and proposes a layout which reveals clusters of frequently co-changed artifacts. Jazayeri *et al.* [11] visualizes software release histories using colors and the third dimension. They do no visualize any coupling relationships between modules.

Girba *et al.* used the notion of history to analyze how changes appear in the software systems [9] and succeeded in visualizing the histories of evolving class hierarchies [10].

Taylor and Munro [16] visualized CVS data with a technique called *revision towers*. Ball and Eick [1] developed visualizations for showing changes that appear in the source code.

Rysselberghe and Demeyer used a simple visualization based on information in version control systems to provide an overview of the evolution of systems [17].

Wu *et al.* described an Evolution Spectrograph [18] that visualizes historical sequences of software releases.

6. CONCLUSION

In this paper we have presented the Evolution Radar, a novel approach to integrate and visualize module-level and file-level logical coupling information. Unlike the previous visualizations in this domain, our approach facilitates an in-depth analysis of logical coupling between entities at different granularity levels. The visualization is useful to answer questions about the evolution of the system, the impact of changes at different levels of abstraction and the need for system restructuring.

We have provided solutions for the problems mentioned in Section 1: The Evolution Radar presents large amounts of information in a condensed way (in the Mozilla examples the number of files was greater than 30'000), guiding the user directly to the files responsible for the modules' logical coupling. The interactive facilities provided by our tool allow the user to inspect/filter entities of interest, to group them and to create ad-hoc visualizations on the fly.

As a case study, we have presented various scenarios of using our visualization technique to support the analysis of more than seven years of evolution of the Mozilla project.

6.1 Future Work

In the future we plan to explore the following research directions:

Structural information. We want to encapsulate structural information like file size, number of methods, lines of code, *etc.* in the Evolution Radar. The challenge in this is finding a way to encapsulate this data in the Radar layout without losing scalability and readability.

Full integration in BugCrawler. In the current implementation the Evolution Radar is an extension of BugCrawler [6]. By merging the two we will be able to navigate from the structural and evolutionary views provided by BugCrawler to the Evolution Radar.

Sliding time window. We want to use the sliding time window approach, instead of the fixed one, to compute the logical coupling measure.

Bug-related information. We want to apply the same visualization technique using the number of shared bugs as a measure for the dependencies. Our hypothesis is that the greater the number of bugs shared by two entities the stronger their dependency is. We will check this hypothesis by comparing the results obtained using the two measures (*i.e.,* logical coupling and bug sharing).

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science foundation for the projects "COSE - Controlling Software Evolution" (SNF Project No. 200021-107584/1), and "NOREX - Network of Reengineering Expertise" (SNF SCOPES Project No. IB7320-110997), and the Hasler Foundation for the project "EvoSpaces - Multi-dimensional navigation spaces for software evolution" (Hasler Foundation Project No. MMI 1976).

7. REFERENCES

[1] T. Ball and S. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.

[2] D. Beyer and A. Noack. Clustering software artifacts based on frequent common changes. In *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC 2005)*. IEEE Computer Society Press, Los Alamitos (CA), 2005.

[3] M. Burch, S. Diehl, and P. Weissgerber. Visual data mining in software archives. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 37–46, New York, NY, USA, 2005. ACM Press.

[4] M. C. Chuah and S. G. Eick. Information rich glyphs for software management data. *IEEE Computer Graphics and Applications*, 18(4):24–29, July 1998.

[5] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, pages 77–86, New York NY, 2003. ACM Press.

[6] M. D'Ambros and M. Lanza. Software bugs and evolution: A visual approach to uncover their relationships. In *Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering)*, pages xxx–xxx. IEEE CS Press, Mar. 2006.

[7] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings International Conference on Software Maintenance (ICSM '98)*, pages 190–198, Los Alamitos CA, 1998. IEEE Computer Society Press.

[8] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 13–23, Los Alamitos CA, 2003. IEEE Computer Society Press.

[9] T. Gîrba, S. Ducasse, and M. Lanza. Yesterday's Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 40–49, Los Alamitos CA, 2004. IEEE Computer Society Press.

[10] T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 2–11, Los Alamitos CA, 2005. IEEE Computer Society.

[11] M. Jazayeri, H. Gall, and C. Riva. Visualizing Software Release Histories: The Use of Color and Third Dimension. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, pages 99–108. IEEE Computer Society Press, 1999.

[12] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of IWPSE 2001 (International Workshop on Principles of Software Evolution)*, pages 37–42, 2001.

[13] M. Lehman and L. Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.

[14] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *Proceedings of SoftVis 2005 (2nd ACM Symposium on Software Visualization)*, pages 67–75, St. Louis, Missouri, USA, May 2005.

[15] J. Ratzinger, M. Fischer, and H. Gall. Improving evolvability through refactoring. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[16] C. Taylor and M. Munro. Revision towers. In *Proceedings 1st International Workshop on Visualizing Software for Understanding and Analysis*, pages 43–50, Los Alamitos CA, 2002. IEEE Computer Society.

[17] F. Van Rysselberghe and S. Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM '04)*, pages 328–337, Los Alamitos CA, Sept. 2004. IEEE Computer Society Press.

[18] J. Wu, R. Holt, and A. Hassan. Exploring software evolution using spectrographs. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 80–89, Los Alamitos CA, Nov. 2004. IEEE Computer Society Press.

[19] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proceedings 1st International Workshop on Mining Software Repositories (MSR 2004)*, pages 2–6, Los Alamitos CA, 2004. IEEE Computer Society Press.

[20] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572, Los Alamitos CA, 2004. IEEE Computer Society Press.