Empir Software Eng (2006) 11: 369–393
DOI 10.1007/s10664-006-9004-6

An empirical study of fine-grained software modifications

Daniel M. German

Published online: 31 May 2006
© Springer Science + Business Media, LLC 2006
Editors: Mark Harman, Bogdan Korel, Panos Linos, Audris Mockus, and Martin Shepperd

Abstract Software is typically improved and modified in small increments (we refer to each of these increments as a modification record—MR). MRs are usually stored in a configuration management or version control system and can be retrieved for analysis. In this study we retrieved the MRs from several mature open software projects. We then concentrated our analysis on those MRs that fix defects and provided heuristics to automatically classify them. We used the information in the MRs to visualize what files are changed at the same time, and who are the people who tend to modify certain files. We argue that these visualizations can be used to understand the development stage of in which a project is at a given time (new features are added, or defects are being fixed), the level of modularization of a project, and how developers might interact between each other and the source code of a system.

Keywords Software evolution · Version control · Software visualization · Software artifacts

1 Introduction

Configuration management systems, and more specifically, version control systems keep records of the modification history of a software project. The logs from these systems track who modifies what, when and what the change was. CVS, the Concurrent Versions System is arguably the most widely used version control management system. In this paper we retrieved and analyzed the CVS logs of several mature projects (PostgreSQL, Apache, Mozilla, GNU gcc, and Evolution) using softChange, a tool that retrieves the history of a project, analyses and enhances the history by finding new relationships within it, and allows users to navigate and visualize this information (German et al., 2004). The first stage was to extract the file revisions in a way similar to (Fischer et al., 2003b). A file revision is a modification to its corresponding file. CVS does not keep track of "transactions," and therefore, it is

D. M. German (✉)
Software Engineering Group, Department of Computer Science,
University of Victoria, Victoria, Canada
e-mail: dmg@uvic.ca

not possible to know which files were committed at the same time by a given author. In this paper we propose an algorithm to recover these transactions, which we call *Modification Records* (MRs). MRs provide a fine-grained view of the evolution of a software product. The majority of the modifications in an MR are to implement small improvements to the source code or to fix defects in the project.

1.1 Research Questions

We started this study by posing some questions:

- What do typical MRs look like? Is it possible to automatically classify MRs into different categories according to their purpose?
- Are MRs different in different stages of software development?
- What is the effect of modularization of the code base on the composition of MRs?
- Do files tend to be modified by only one developer?
- Can we infer some type of social network among developers from the modification patterns of a software project?
- How can we visualize files and their relationship to MRs to answer some of the above questions?

1.2 Organization

This paper is divided as follows: we continue in Section 2 by describing our methodology. In Section 3 we analyze modification records and propose some metrics and visualizations of MRs and their authors. We then continue with a discussion of our results. In Section 6 we survey previous and related work in this area. We conclude with a summary of this paper and propose future work.

2 Methodology

The first part of our research was the retrieval of the historical information from CVS and the rebuilding of its MRs. As described in the previous section, CVS does not keep track of which files are modified together and therefore, the first task was to rebuild this information. softChange uses a heuristic that is based on a sliding window algorithm to rebuild MRs based on its component file revisions. This algorithm takes two parameters as input: the maximum length of time that an MR can last δ_{max}, and the maximum distance in time between two file revisions τ_{max}. This algorithm is depicted in Fig. 1. Briefly, a file revision is included in a given MR if a) all the file revisions in the MR and the candidate file revision were created by the same author and have the same log (a comment added by the developer when the file revisions are committed); b) the candidate file revision is at most τ_{max}seconds apart from at least one file revision in the MR; and c) the addition of the candidate file revision to the MR keeps the MR at most δ_{max} seconds long.

Most MRs take only few seconds to complete, but some tend to be rather longer. There are several factors that affect the duration of an MR. First, the size and number of files that compose the MR; second, the bandwidth available between the developer's computer and the CVS server (a slow link will slow down the time required to do the commit); and third, the load of the CVS server. We tested the extraction for different parameters of τ_{max} and δ_{max} in two different CVS repos-

```
// front(List) removes the front of the list
// top(List) and last(List)
//    query the corresponding elements of the list
// Initialize set of all MRs to empty
MRS = ∅
for each A in Authors do
    List = Revisions by A ordered by date
    do
        MR.list = {front(List)}
        MR.sTime = time(MR.list_1)
        while first(List).time − MR.sTime ≤ δ_max ∧
                first(List).time−
                    last(MR.list).time ≤ τ_max ∧
                first(List).log = last(MR.list).log ∧
                first(List).file ∉ MR.list do
                queue(MR.list, front(List))
        od
        MRS = MRS ⋃ {MR}
    until List ≠ ∅
od
```

Fig. 1 Algorithm for the recovery of MRs

itories (Apache 1.2 and Evolution). The results are shown in Figs. 2 and 3. In Fig. 2 τ_{max} was varied from 1 to 600 s, while δ_{max} was fixed at 600 s. In Fig. 3 δ_{max} was varied from 1 to 600 s, while τ_{max} was fixed at 45 s. The vertical axis is the proportion of MRs rebuilt for a particular parameter value. For example, in Fig. 2 the number of MRs varies by less than 1% when $\tau_{max} > 25$, and the number of MRs increases as much as 20% when $\tau_{max} = 1$. δ_{max} has a different effect: when it is equal to 1 s, MRs increase by almost 40%, but their number remains almost constant for both projects for $\delta_{max} > 50$.

These graphs suggest that $\delta_{max} = \infty$ and $\tau_{max} = \infty$ will yield useful values. We have found, however, that the algorithm depends greatly on how descriptive the logs of CVS commits are. If all the comments are empty then it will be more difficult to distinguish between two different MRs that are submitted by the same author within few minutes of each other. The average length of comments is 135 and 337 characters for Apache 1.2 and Evolution, respectively. In our experiments we used $\tau_{max} = 45s$ and $\delta_{max} = 600s$ for the extraction of MRs.

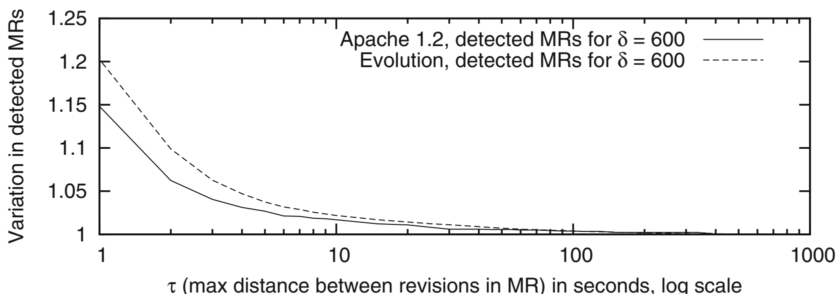

Fig. 2 Variability in the number of recovered MRs when τ_{max} is varied

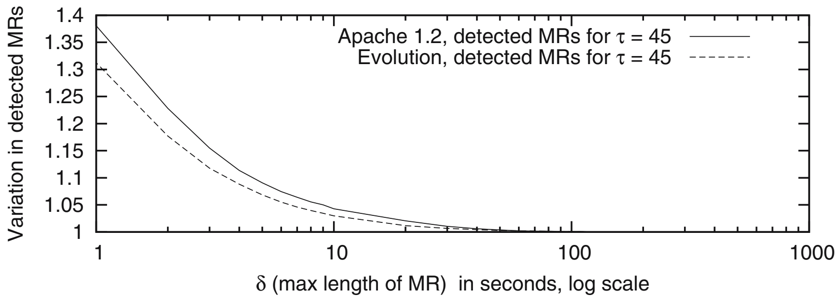

Fig. 3 Variability in the number of recovered MRs when δ_{max} is varied

MRs record all types of activity: changes to the source code, documentation, internationalization, etc. We decided to concentrate our attention in the evolution of the code base, hence, we looked into MRs that included source code files, which we call codeMRs. A codeMR is an MR that contains at least one source code file revision. For this research, we used the CVS repositories of the following projects:

- Apache. A Web server. Its development has been divided into several CVS repositories, one for each major version. We used the repositories of Apache 1.2 and Apache 1.3.
- Evolution. A mail client for Unix similar in functionality to Microsoft Outlook.
- GNU gcc. The GNU multi-platform, multi-language compiler.
- Mozilla. A Web browser.
- PostgreSQL. A relational database.

All these projects are open source, stable, several years old, and have a large user base. Table 1 shows some statistics of each of them. We obtained copies of the CVS repositories of Apache and Evolution (except its internationalization data), and the rest were remotely mined using the CVS protocol. The resulting MRs were loaded into a relational database. A more detailed discussion of the extraction stage, including the schema of this database is presented in (German, 2004b).

For four projects (Apache 1.2, Apache 1.3, Evolution and PostgreSQL) we proceeded to analyze their history in more detail. First, we instantiated every revision of every source code file and its corresponding “clean” version. A clean version of a source code file is computed using the following algorithm:

1. Remove all the comments and empty lines
2. Reformat the resulting source code

Table 1 Statistics of the projects used in this paper

Project	First date	Last date	Files	Revisions	MRs	codeMRs	Authors
Apache 1.2	1996-01-14	1999-08-30	428	4,646	1,839	1,073	18
Apache 1.3	1996-01-14	2004-11-14	1,293	20,794	8,021	3,983	60
PostgreSQL	1996-07-06	2004-09-18	5,579	91,740	20,330	11,023	27
Evolution	1998-01-12	2003-11-18	5,127	81,874	18,216	14,032	148
Mozilla	1998-03-28	2003-10-04	35,229	452,738	117,554	85,618	528
GNU gcc	1997-08-11	2004-02-18	24,463	485,930	58,639		214

If the clean version of a revision was identical to the clean version of its preceding revision then the file had changed only in its comments or its white space and its source code was not changed.

3 Analysis of Source Code Modification Records

What do source code modification records (codeMRs) look like? We started with the assumption that there exist different types of codeMRs, which reflect the type of activity that the developer is completing. Based on our observations we have identified six types of codeMRs:

- Functionality improvement, e.g., implementation of new features.
- Defect-fixing.
- Architectural evolution and refactoring, e.g., a major change in APIs or the reorganization of the code base.
- Relocating code, e.g., a file is placed in a new directory, or a function is moved from one file to another.
- Documentation. Since we are concentrating only on source code, this means changes to the comments within files.
- Branch-merging, e.g., code is merged from a branch or into a branch. Branches are used for different purposes in software development: they might record experimental code that cannot be committed immediately to the main development branch of the system (the HEAD). Some branches are never "merged" to the HEAD. Sometimes branches record alternate versions of the system that evolve independently of the HEAD of the system.

This list might not be comprehensive, but we believe these are the most common types of changes present in MRs. Furthermore, these types are not mutually exclusive: for example, an MR can include files with major changes in their documentation and those files might be moved to a different location at the same time. Documentation and architectural evolution MRs tend to include a large number of files. For example, we have discovered MRs composed of several hundred files in which each file has only been modified in its comments (in one instance the license of the product changed; in another, the name of the company changed). Architectural changes might involve moving large amounts of code from one place to another, or changing APIs of important components that would require any file using that function to be changed.

The main problem we face is: can we classify automatically codeMRs into sets that are somehow similar to the previous categories? We were able to identify heuristics to classify codeMRs into the following three different types:

- **bugzillaMRs** are codeMRs that correspond to an explicit defect fix as recorded by Bugzilla. Usually, if the MR fixes a bug, the developer will record the defect number in the log of the MR. Unfortunately, of the projects analyzed herein, only Evolution and Apache have a Bugzilla site. In both projects Bugzilla defect numbers are usually preceded by a # or the word *bug*. If an MR log matches one of the following regular expressions, it is considered to be a bugzillaMR:
  - bugs?[\n\t]+[0–9]+,
  - #[\n\t]*[0–9]+

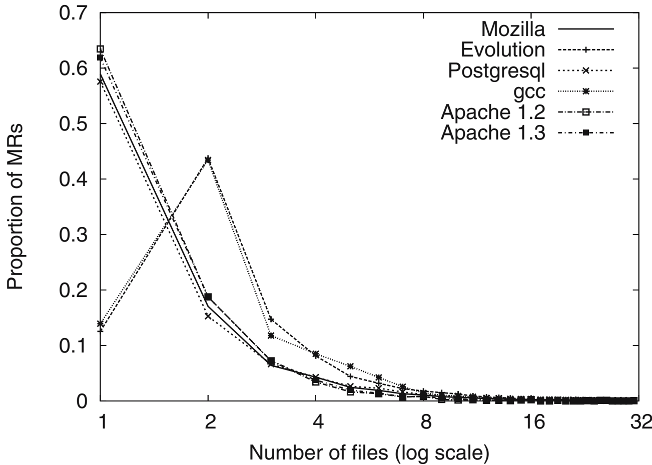

Fig. 4 Proportion of MRs with a given number of files for various projects. Most MRs are composed of one or two files

- **fixDefectMRs**. We have found that the developers of mature projects record a good explanation of the change in the CVS commit log. From our observations two words are very useful to detect if a change is a defect fix: *bug* and *fix* (including their derivatives: *bugs*, *fixed and fixes*). A codeMR that contains any of these words is marked as a fixDefectMR.
- **commentMRs** are codeMRs in which every one of its source code files was modified only in its comments. For a given MR, if every one of its clean file revisions was identical to the its previous clean revision, then the MR was labeled as a commentMR. commentMRs do not change the functionality of the system.

How do these categories relate to the ones previously presented? bugzillaMRs and fixDefectMRs are subsets of the defect-fixing type of MRs, while commentMRs are a subset of the documentation type. commentMRs are relatively uncommon: in Evolution less than 3% of codeMRs are commentMRs while in Apache 1.2 the proportion is 4%, and 5% for Apache 1.3. fixDefectMRs are more common (they are approximately 25% of all code MRs). bugzillaMRs varied the most: Apache 1.2 had 4% of its codeMRs labeled as a bugzillaMR, while Apache 1.3 had 2% and Evolution 11%. Another important issue is that the union of these three categories corresponds to a subset of all codeMRs. According to the projects we analyzed, these three categories cover only between 30 to 40% of all the codeMRs. If we exclude commentMRs, these heuristics only discover MRs that fix defects. The rest of the codeMRs will correspond to functionality improvements, architectural evolution and refactoring, code relocation, etc., and those defect-fixing MRs that are not detected by these heuristics.

How good is the recall of these classification heuristics? In the detection of commentMRs we use the Unix indent program to reformat source code (indent was designed for C). Based on our observations this heuristic performs well, but we have not conducted a formal evaluation of its accuracy.

To evaluate the precision of our heuristics for detecting bugzillaMRs and fixDefectMRs we randomly sampled 100 detected bugzillaMRs from Evolution, and

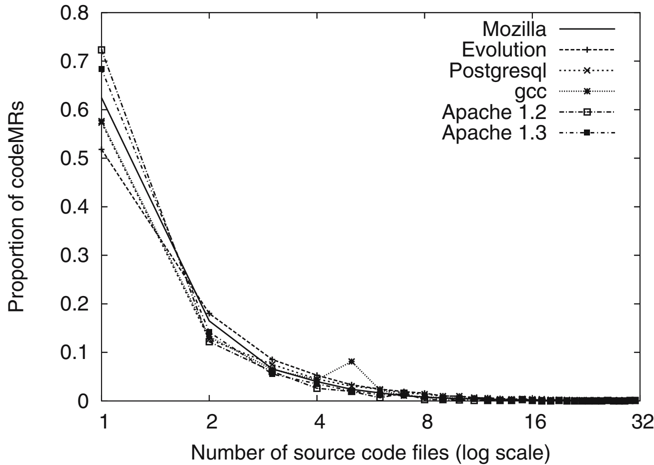

Fig. 5 Proportion of codeMRs with a given number of source code files for various projects

100 detected fixDefectMRs from Apache1.2. Ninety-nine bugzillaMRs were correctly classified, and all of the fixDefectMR appeared to have been correctly classified (in the opinion of the author—determining what is a "fix" is very subjective). Should all bugzillaMRs be classified as fixDefectMRs? We would expect the answer to be yes, were Bugzilla intended to track only bug fixes. We have found, however, that a very small number of bugzillaMRs are the implementation of new features (they fall into the "wish" category in Bugzilla). For Evolution only 12% of bugzillaMRs were not classified as fixDefectMRs. An example of a bugzillaMR that was not classified as a fixDefectMRs has the log entry: 'Add accelerators. [#10068].' Is adding accelerators (an accelerator is a predefined key that triggers a menu action) a defect fix? Some could argue that it is a defect fix, while others can argue that it is a new feature. Without clear specifications (which most open source projects lack) there is no clear answer to this question.

3.1 Number of Files in MRs

We looked in detail into the number of files in MRs. We discovered that in all the projects most MRs tend to contain very few files. Figure 4 shows the distribution of the number of files in MRs for the projects Apache 1.2, Apache 1.3, Evolution, GNU gcc, PostgreSQL and Mozilla.

The plot only shows only up to 32 files. There are very few larger MRs (for example, in Evolution we detected an MR which included 650 files, and in Mozilla one that included 5,838 files). Note that two distinctly different curves are obtained: one comprising the results for Apache 1.2 and 1.3, PostgreSQL and Mozilla; the other including the results for Evolution and GNU gcc. This effect was interesting enough to explore further. We discovered that the use of ChangeLog files (the file ChangeLog is part of the GNU standard and includes an explanation of what has been changed by who and when) accounted for this sharp difference. Evolution and GNU gcc use ChangeLogs, and almost every MR that includes two or more files includes also a modification to a ChangeLog file. Mozilla and PostgreSQL do not use them. When

Table 2 Average and standard deviation in the number of source code files in codeMRs for various projects

Project	codeMRs	Avg.	Std. dev.
Apache 1.2	1,073	2.19	5.32
Apache 1.3	3,983	2.45	7.07
PostgreSQL	11,023	5.21	27.11
Evolution	14,032	3.41	10.08
Mozilla	117,554	3.00	13.72

ChangeLogs are not taken into account the curves of all the projects look remarkably similar.

Given that our interest was centered on software development, we proceeded to count only source code files in codeMRs. Figure 5 shows the accumulated distribution of the number of source code files in codeMRs, and Table 2 shows the average and the standard deviation for these projects. The distributions are very similar; their average is relatively small (Apache 1.2 has the smallest, and PostgreSQL the largest), but their standard deviation is less uniform (from 5.32 in Apache to 27.11 in PostgreSQL). One explanation for this effect is that PostgreSQL developers tend to commit MRs that implement more complex functionality, while the Apache developers tend to be more conservative in their changes. Further research is needed to explain this behavior.

We inspected the codeMRs of Evolution. Figure 6 shows the distribution of the size of the three types of detected MRs (including the total number of codeMRs for reference purposes) and Table 3 shows their main statistics. We can make several observations about them: most commentMRs are composed of changes to one file (80% have one file, which means its changes are not documented in the ChangeLog) or many (5% of commentMRs contain at least 16 files, and the largest has 256). One can argue that developers who comment a source code file do not feel the need to add a comment to the ChangeLog explaining what they have done. Large commentMRs tend to be changes to the comment at the top of each file, which usually contains a copyright

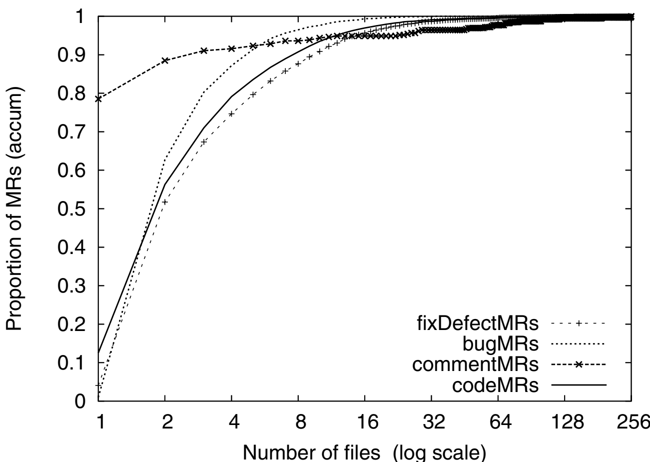

Fig. 6 Distribution of the total number of files (including non-source code) in different types of MRs for Evolution

Table 3 MR statistics for evolution

Type	Total	Avg. size	Std. dev.
All MRs	18,216	4.50	13.64
codeMRs	14,032	3.41	10.08
bugzillaMRs	1,645	1.95	2.25
fixDefectMRs	3,746	3.70	10.08
commentMRs	391	4.85	20.62

The size is the number of source code files in the MR.

clause. We found that the commentMRs with the largest numbers of files corresponded to a change in the copyright (either from one copyright holder to another, or updating its year).

It is unusual for fixDefectMRs and bugzillaMRs to modify just one file. We found that 95% of bugzillaMRs and 90% of fixDefectMRs included a modification to a ChangeLog that documented the change. As depicted in Table 3, bugzillaMRs had the smallest average size (3.07 files) and the smallest standard deviation (2.54).

We can conclude that, for Evolution, the MRs that are labeled as bug fixes contain few files, and MRs that only modify comments in the source code are likely to contain either just one file or a large number of files.

Do these observations extend to other projects? We analyzed the MRs of Apache 1.3. Table 4 shows the statistics of the different types of MRs of the project. Figure 7 shows the corresponding plot for the project. Considering that Apache does not use ChangeLogs, the average size of a codeMR is very similar to the average size of a codeMR in Evolution. As in Evolution, most of Apache's commentMRs have only one file, but many of these are very large. On the other hand, bugzillaMRs in Apache are contain more files than in Evolution. Is this because bugzillaMRs are more complex to fix in Apache than in Evolution? Is it because only more difficult defects are recorded in Bugzilla for Apache, or because Apache developers only record defect numbers in more complex defect fixes? Further research is needed to answer these questions.

3.2 Modification Coupling of Files in codeMRs

In Fischer et al. (2003a), Fisher and Gall argued that historical modification logs can be used to detect a coupling relationship between two files: if two files are modified at the same time, then these two files are related. We decided to analyze when and how the same two files were modified together. For this purpose we defined three metrics:

- *Frequency of modification*. The frequency of modification of a file A $Frequency(A)$ is the number of MRs in which the file A is modified. For a pair

Table 4 MR statistics for Apache 1.3

Type	Total	Avg. size	Std. dev.
All MRs	8,021	2.59	7.30
codeMRs	3,983	2.45	7.07
bugzillaMRs	152	2.47	5.27
fixDefectMRs	1,031	1.84	5.51
commentMRs	209	4.08	12.04

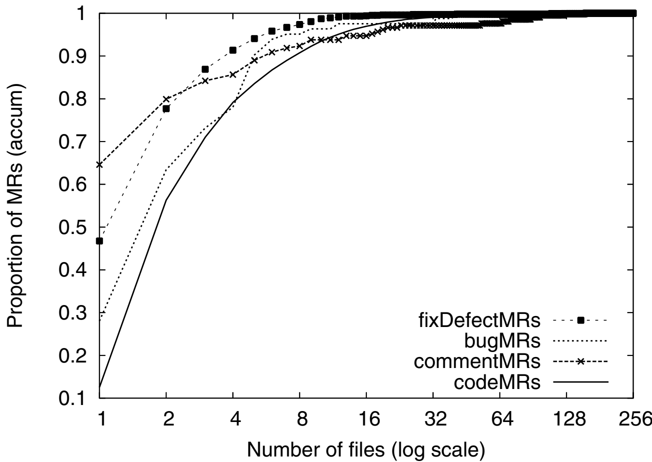

Fig. 7 Distribution of the total number of files (including non-source code) in different types of MRs for Apache 1.3

of files A, B, *Frequency*(A, B) corresponds to the number of MRs that include A and B.

- *Modification coupling*. The modification coupling between two files A and B is the likelihood that if file A is modified, then file B is also modified. Formally, we define the modification coupling between two files A and B as:

$$M(A, B) = \frac{Frequency(A, B)}{Frequency(A)}$$

M is, therefore, non-reflexive.

- *Modification neighbors*. The modification neighbors of file A are all the files that have been modified at the same time as A. Formally, if $R(A)$ is the set of all MRs in which A has been modified, its modification neighbors *Neighbors*(A) is the union of the files modified in the MRs in $R(A)$.

If two files have very high modification coupling one would expect that they are related somehow. This relationship can be very explicit (if one changes the prototype of a function, one has to change every function call), but sometimes it can be more subtle (for example, the format of a configuration file has changed, and the function that reads the file and the one that writes the file have to be updated, even though there is no explicit function call between both).

The size of the modification neighbors can be used as an indicator of how related two files might be. Some files will tend to be modified with many others. Not surprisingly, in Evolution ChangeLogs are the files with the largest set of modification neighbors. We found that the source code file with the largest set of modification neighbors in Evolution was `mail/mail/callbacks.c` (it was also the most frequently modified source code file); this contains the main execution loop of the application, and is responsible, as its name implies, for responding to events and triggering the corresponding callbacks. Whenever a new callback is created this file has to be updated (usually because a new feature has been added to the system).

As it was explained earlier, commentMRs do not contain changes to source code. The relationship between files that are modified together in a commentMR is not very useful. If two files are modified together in a commentMR, this relationship is not as important as when they are modified, for example, during a bugzillaMR. For instance, if an MR modified the copyright of 5,000 files, are they truly related except by the fact that all shared a copyright notice? Branch-merging MRs pose a similar problem: they contain several logical changes in one MR, and, in our opinion, dilute the modification coupling relationship of their component files, as they incorrectly (for the purpose of modification coupling) relate files that belong to two or more smaller operations.

Because we wanted to focus on MRs that showed a higher level of modification coupling between the files that composed them, we chose to ignore MRs that contained a large number of files or MRs that we knew contained files that were only weakly related. As a result we selected a subset of codeMRs based upon the following rules:

1. Select all the codeMRs that contain at least two source code files.
2. Eliminate commentMRs.
3. Eliminate codeMRs in branches. Branches in CVS pose difficult problems for researchers (Fischer et al., 2003b). One of the reasons is that it is not explicit when code from the main trunk of the development tree (the HEAD) is committed to the branch, or when code from the branch is committed back into the HEAD. If branches are not properly processed, a change to a file might be taken into account twice: once when it goes into the branch, and once when it goes back into the HEAD. Furthermore, branch-merging into the HEAD will likely include many files (they are all related, but we are interested in the finer-grained relationships of each MR that was committed to the branch, rather than the entire one). Because branch-merging detection is expensive and error-prone we decided not to take into consideration any revision committed to a branch.
4. Eliminate codeMRs that contain a relatively large number of file revisions. This step attempts to remove from our analysis MRs that are more likely to be architectural, or branch-merging codeMRs (we kept codeMRs with at most 20 files).
5. Eliminate codeMRs that contain files that were deleted in that MR. Unfortunately CVS does not support moving files. The user has to delete and create a new file in a new location. By eliminating these MRs we expected to avoid some code reorganization MRs.
6. Eliminate codeMRs that include the first version of a file. It is frequent that the first revision of the file is the result of importing a group of files into the project, or the result of moving a file (see previous item). We decided to avoid any MR that included a first version of a file.
7. Add all bugzillaMRs (if the project maintains a Bugzilla database), because we have explicit knowledge that these files are related to fixing a defect. One risk of adding all bugzillaMRs is that some of them might complete more than one task.

It is possible that the modification neighbors of a file change as the project evolves. To facilitate our analysis and to avoid trends early in the development of the product, we selected only MRs from one year (2002 for Evolution and PostgreSQL, and 1998 for Apache 1.3—1998 was one of its most active years). Table 5 shows statistics of the three projects. It only includes codeMRs with at least two source code files.

Table 5 Statistics of the working set of three projects (except for *all MRs*, each MR contains at least two source code files)

Project	Year	All MRs	codeMRs	HEAD codeMRs	Working set
Evolution	2002	2,707	1,049	945	823
Apache 1.3	1998	2,301	1,012	327	278
PostgreSQL	2002	2,863	669	644	504

We believe that the modification coupling depends heavily on the programming language of choice. The projects we analyzed were written in C and C++. In C/C++ prototypes, class definitions and constants are usually placed in a `.h file`. If a prototype in the `.h file` is changed, one would expect that the corresponding `.c file` would be changed accordingly. We were, therefore, interested to verify the following hypothesis:

- For most `.h files` and their corresponding `.c file` (`file.h` and `file.c`), $M(file.h, file.c)$ will tend to be close to one (i.e., if `file.h` is modified, `file.c` is almost always modified).

We proceeded to compute the modification coupling for all `.h files` and their corresponding `.c file`. Figure 8 shows the distribution of the frequency of modification couplings for every pair of candidate files. The horizontal axis corresponds to all the possible modification coupling values, while the vertical axis is the proportion of `.h files` with less or equal that modification coupling. This plot clearly shows that for Evolution and PostgreSQL most `.h files` are modified with their corresponding `.c file`. For example, in Evolution 15% of `.h files` have less or equal to 0.95 modification coupling with their corresponding `.c files` (hence 85% have modification coupling > 0.95). PostgreSQL has a similar distribution, with 75% of all `.h files` with > 0.95 modification coupling. In Apache 1.3 this effect is less pronounced: 50% of `.h files` have a modification coupling < 0.5. We found that in most cases, these changes are to constants and macros that do not require changes to the corresponding `.c file`.

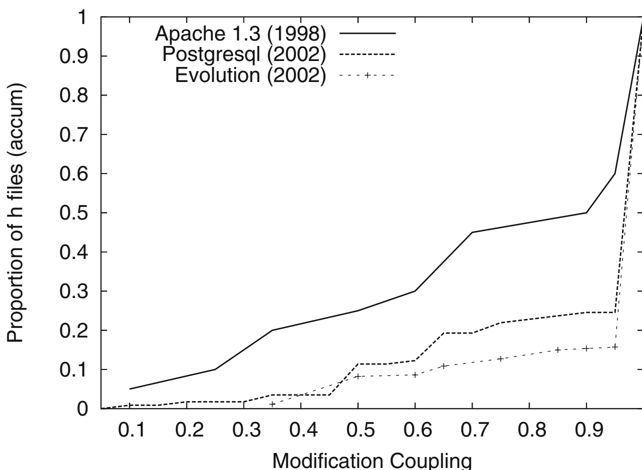

Fig. 8 Proportion of `.h files` with a given modification coupling with their corresponding `.c file`

Other important questions are whether files are changed usually in groups, and whether changes to files tend to be localized to the same module (we define a module as each of the main directories of the source code). We wanted to verify the following hypotheses:

- *A file is usually modified with the same files*, and
- *Most MRs are composed of files that belong to the same module.*

To test these hypotheses we created a coupling graph. A coupling graph is an undirected graph that gives a visual overview of how files are modified together during a given period. Each node of a modification graph corresponds to a file that has been modified. If two files, A and B have been modified together in the same MR, then an arc is created between them. In the visual representation of the graph the arc's width is equal to $Frequency(A, B)$ (the number of times they have been modified together). To improve the layout of the graph nodes are coloured and clustered according to the module they belong to.

The graph for the entire working set was too large and busy to include here. Instead we decided to concentrate on smaller periods. There were two interesting periods around a major release: for Evolution it was 2002/11/07, when version 1.2.0 was released (a major stable release). The month of October 2002 was spent making sure that there were no errors in the version, while most of November was spent adding features for the next new unstable release. We will refer to October 2002 as the *Maintenance* period, and to November 2002 as the *Improvement* period. Figure 9 shows the coupling graph for the maintenance period and Fig. 10 shows the coupling graph for the improvement period.

We can make the following observations:

- The maintenance period has significantly fewer MRs than the improvement period. Unfortunately we do not know much time developers spend during each period. We can only speculate that they spend similar times during both periods, and, as expected, the number of MR resulting from bug fixing is significantly smaller than adding new functionality to the system.
- The graphs tend to be composed of small disconnected subgraphs, or clusters of nodes interconnected by few edges. This suggests that a file is modified with few other files. There are, however, files that have a large number of modification neighbors. It is also easy to spot files that are frequently modified together (the thicker the line, the more frequently they have been changed together). During the maintenance period most files were modified together only once, but during the improvement period several ones were modified together multiple times. During the maintenance period the average number of modifications per file was 1.18, and some files were modified at most three times. For the improvement period the average was 1.98 the maximum frequency was 9.
- The modularization of the project has a profound impact in the disjointedness of the different subgraphs. During the maintenance period there is no MR that spans two modules, and in the improvement period only two modules are interconnected by a total of three files. It is believed that the success of an open source project depends on the ability of its maintainers to divide it into small parts in which contributors can work with minimal communication between each other and with minimal impact on the work of others (Lerner and Triole, 2000). This division of work seems to be based on the modularization of the project.

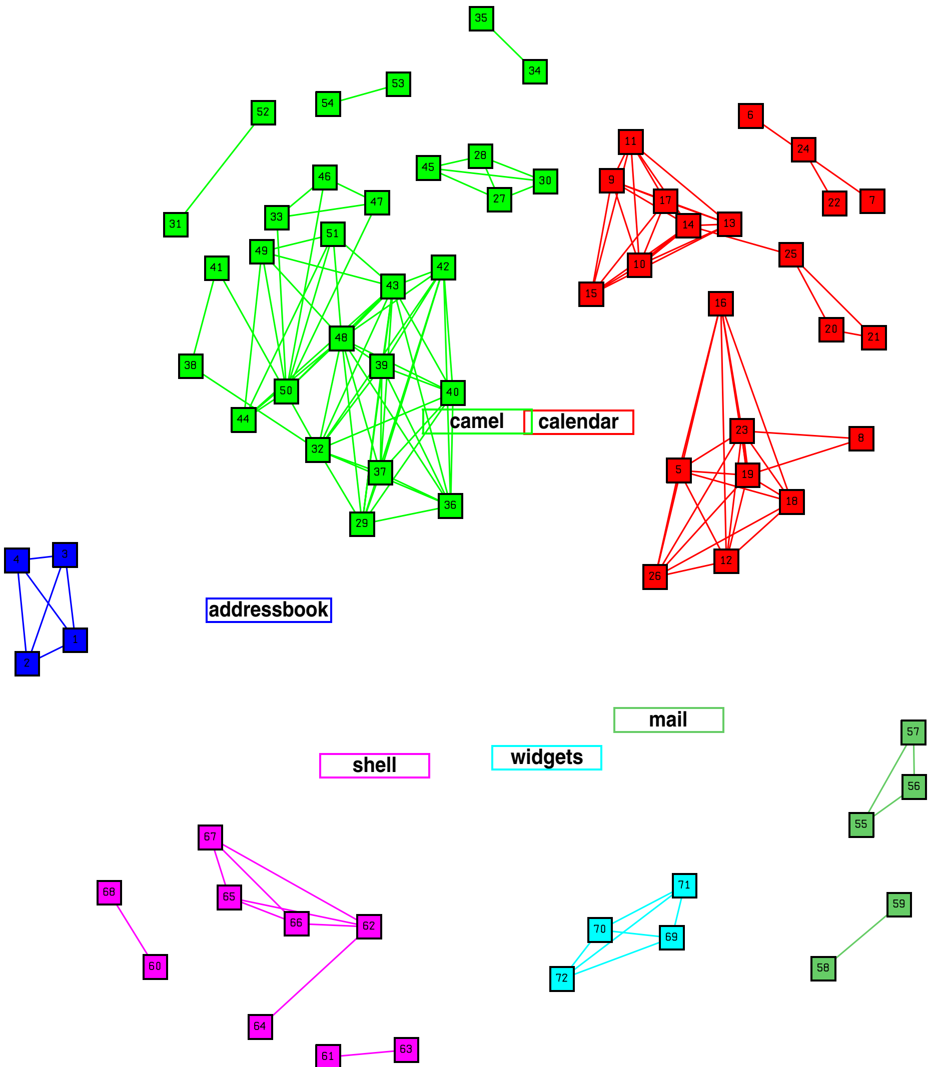

Fig. 9 Modification coupling graph for *Maintenance* period of Evolution. *Squares* represent files and two files are connected if they were modified together

3.3 Authorship

We were also interested in understanding how authors are related to modification coupling. We wanted to answer the following questions:

- How many people tend to modify a given file?
- How many authors contribute to a given module?
- Can we infer some type of social network from the modification patterns of a software project?

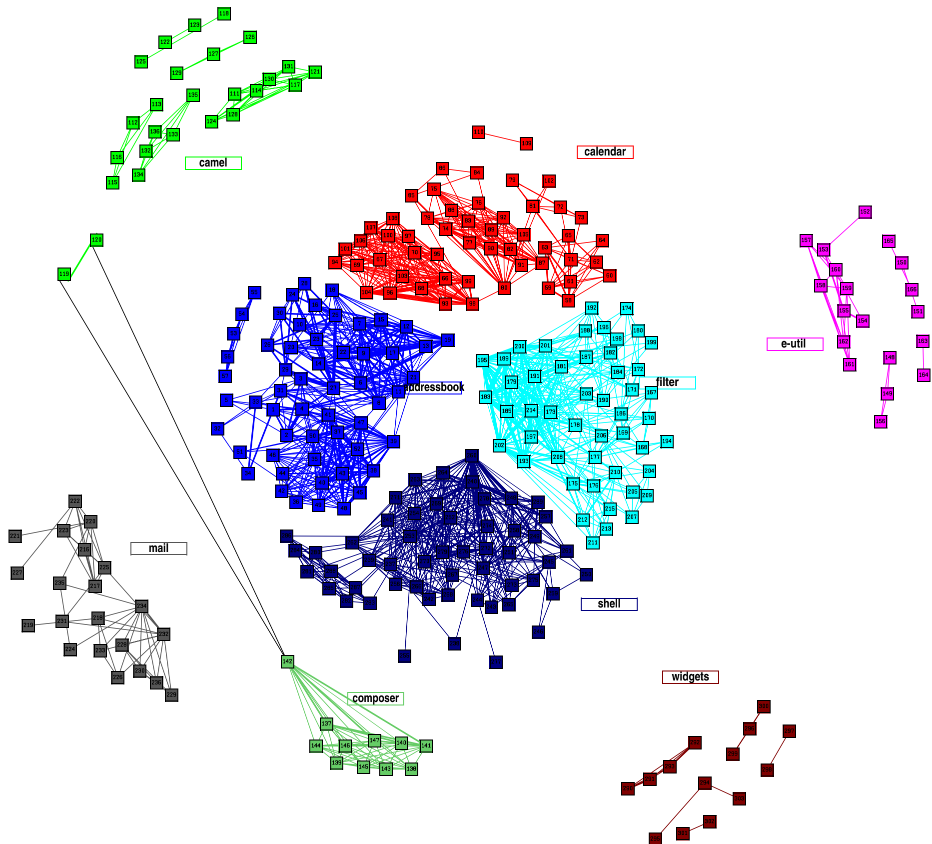

Fig. 10 Modification coupling graph for *Improvement* period for Evolution

We proceed to define an authorship graph. In an authorship graph there were two types of nodes: files, and authors. There exists a node for each file that has been modified, and one per each author who has submitted an MR. An edge is created between a file F and an author *A* if A had modified the file F. In the visual representation of the graph files are depicted as squares and authors as ovals. File nodes are also clustered and coloured according to the module they belong to. Finally, the width of an arc is proportional to the number of times an author has modified a given file. Figure 11 shows the authorship graph for the maintenance period, while Fig. 12 corresponds to the improvement period. We can make the following observations about these graphs:

- Most files are modified (“owned”) by one individual. The graph is very helpful in identifying who this individual is.
- Most individuals tend to concentrate their work in one or two modules.
- The files in some modules are split into few clusters where each cluster is primarily modified by one author.
- The files that are modified by more than one author might be “interfaces” between two different parts of the system. They might contain code that connects the functionality of one module with the other. Further research is needed to very this assertion.

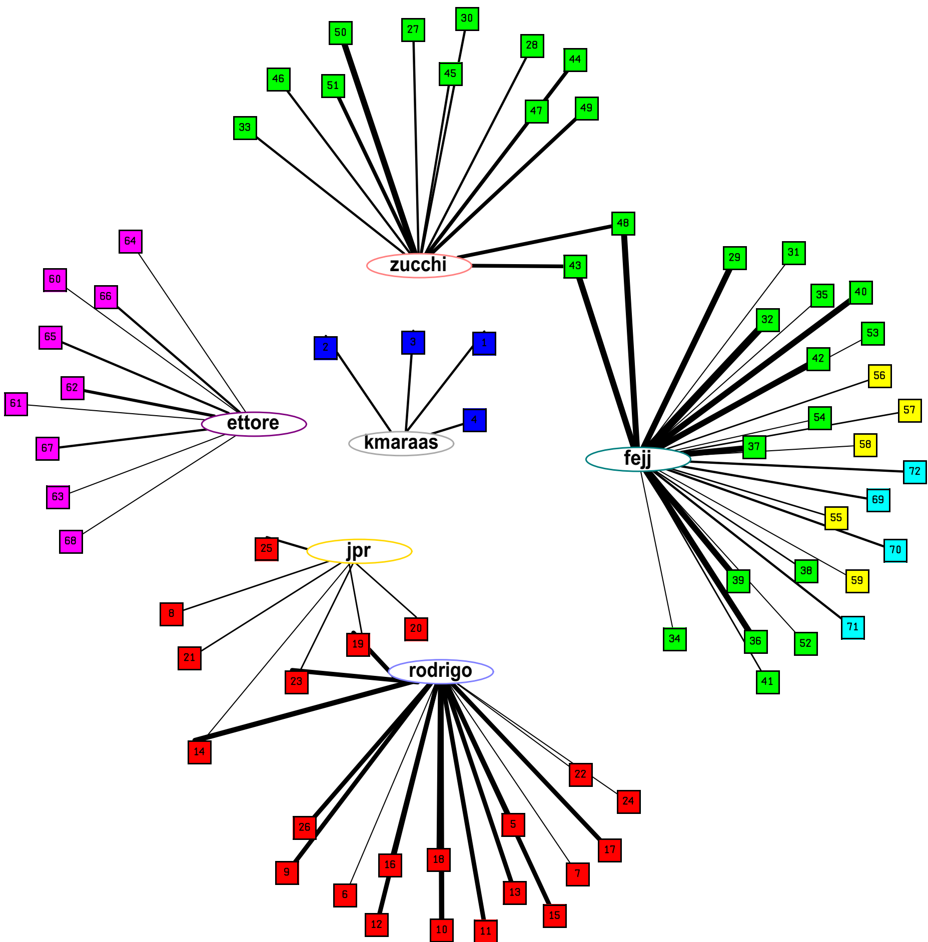

Fig. 11 Authorship graph during *Maintenance* period of Evolution. *Ovals* represent authors that are connected to the files they have modified (represented as *squares*)

In order to understand the effect of modularization in these graphs, the files of each module were collapsed into a single node. If an author modified a file in that module then an arc was created. The result can be seen in Fig. 13. Authors tend to modify code in few modules (some in one only). The information of this graph can be useful for developers; for example, it can help them by making them aware of who are the people who tend to work in a given part of the system.

We proceeded to do a similar analysis for PostgreSQL around the release of version 7.2 (2002/02/04). Figures 14 and 15 show the modification coupling graphs for the months before and after the release, and Fig. 16 the authorship graph for the *Maintenance* period. The results are similar to those in Evolution: the maintenance period has significantly less MRs than the improvement period; also, files tend to be modified in clusters and changes tend to be localized in one directory. There is two modules that are highly connected in these two graphs (modules in the center and the

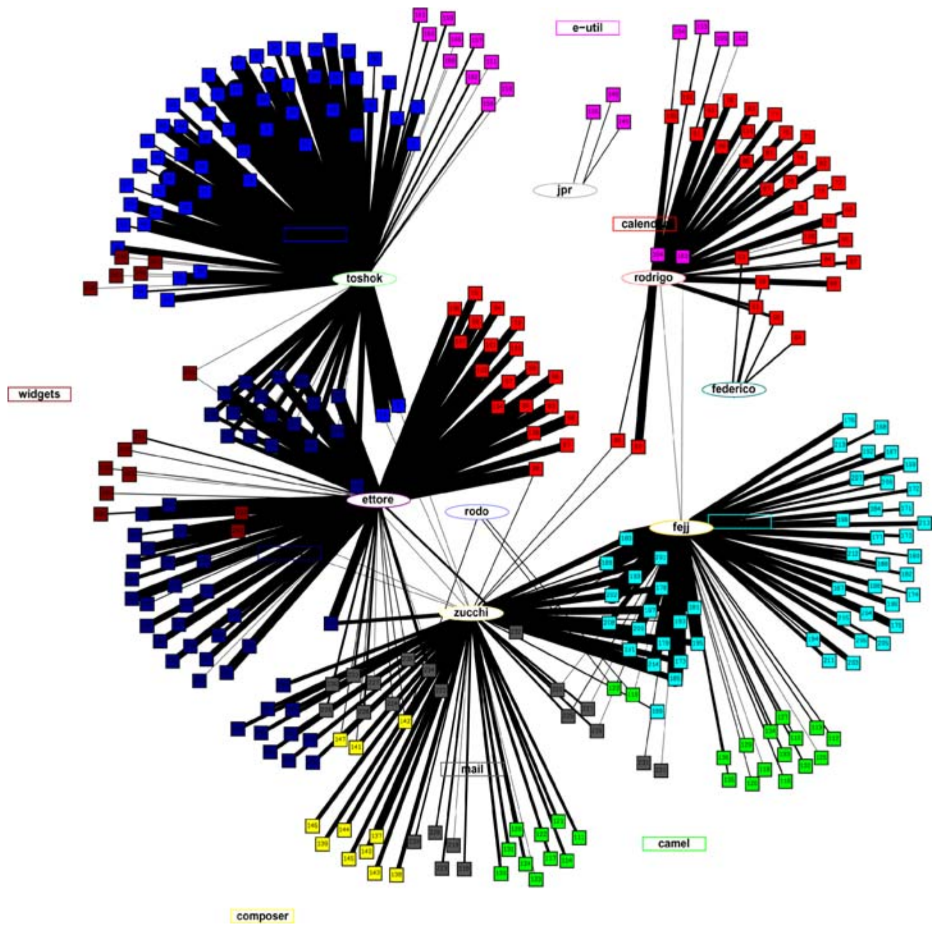

Fig. 12 Authorship graph during *Improvement* period

bottom left in 14, and the two modules in the center of 15). One of the them is the *include* module, where the `.h files` reside. It is interesting to see that in PostgreSQL all include files are located in their own module, while in Evolution they are within the module of their corresponding `.c files`.

4 Discussion

Our original goal was to classify automatically codeMRs based on the type of activity that they reflect: functional improvement, defect fixing, architectural Evolution and refactoring, documentation, etc. One important question is how our classification of codeMRs into bugzillaMRs, commentMRs, and fixDefectMRs relate to these categories of activity. commentMRs are clearly of the documentation type, while bugzillaMRs and fixDefectMRs are most likely defect-fixing activities. An important issue is that we have classified only a fraction of the original codeMRs: the sum of bugzillaMRs,

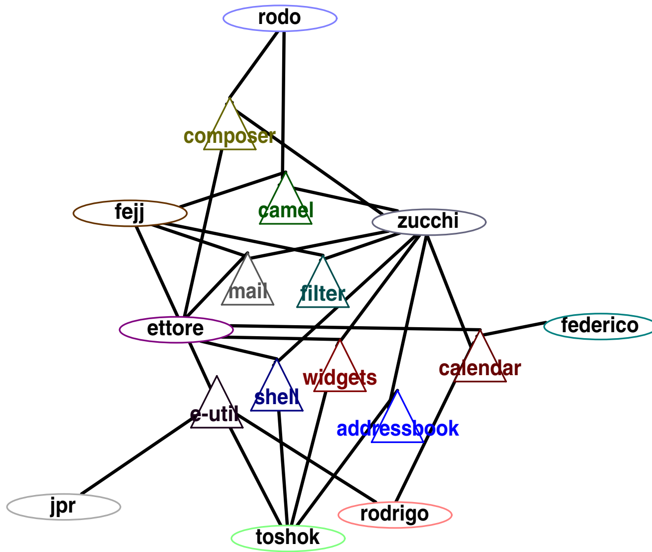

Fig. 13 Collapsed authorship graph for *Improvement* period. A *triangle* represents a module and an *oval* represents an author. If an author has modified at least one file in a module they are connected

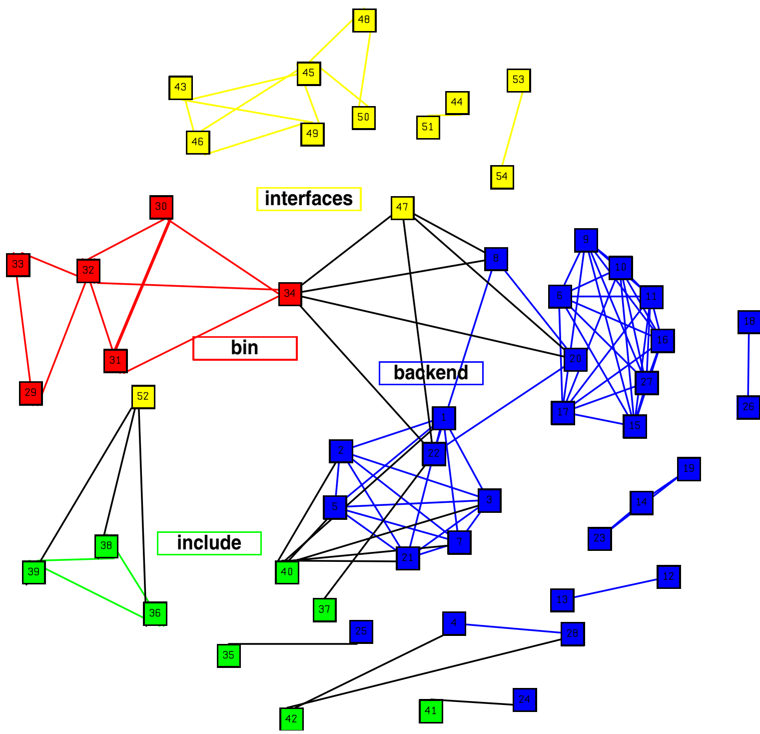

Fig. 14 Modification coupling graph for *Maintenance* period of PostgreSQL

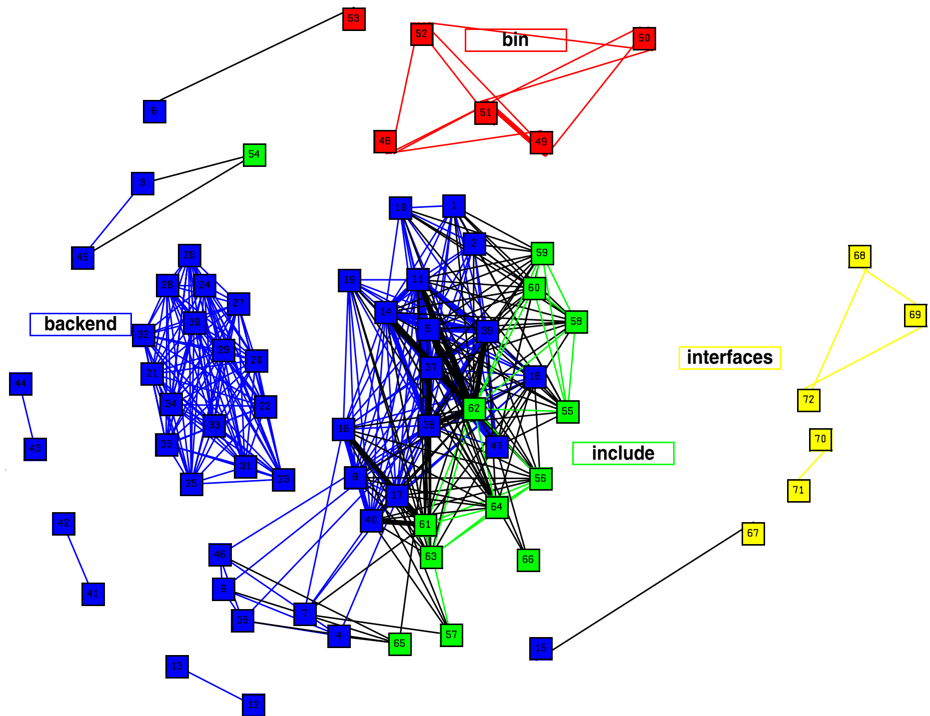

Fig. 15 Modification coupling graph for *Improvement* period of PostgreSQL

commentMRs and fixDefectMRs corresponds to approximately 1/3 of all codeMRs. The rest of the codeMRs comprises a large set including functional improvements, architectural Evolution and refactoring, and codeMRs missed by our heuristics (we will refer to this set as otherMRs). Further work is needed to determine what otherMRs look like and if they can be automatically analyzed and divided into different subsets.

As we mentioned in Section 3, some codeMRs involve more than one activity, and we do not know how prevalent these are, although we suspect that their proportion will change from project to project, according to the habits of their developers. For example, some fixDefectMRs might not be just a defect fix, and might include new functionality. Also, using the heuristics described in this paper, documentation MRs are not detected as commentMRs if they also modified some source code. We are also unable to detect a defect-fixing codeMR that was not properly tagged under Bugzilla or does not include any of the keywords "bug" or "fix."

Despite these limitations, even if our heuristics are only able to find subsets of their corresponding types of MRs, their high level of precision makes them useful. For instance, we plan to use machine learning to find defect-fixing codeMRs that currently are not being detected. We can use the bugzillaMRs and fixDefectMRs as a training set, and then attempt to discover other similar changes in the history of a project. An added advantage of this approach is that bugzillaMRs and fixDefectMRs are likely to show a stronger interrelation between their component entities (files, classes, methods or functions) than the average codeMR. Several techniques use these

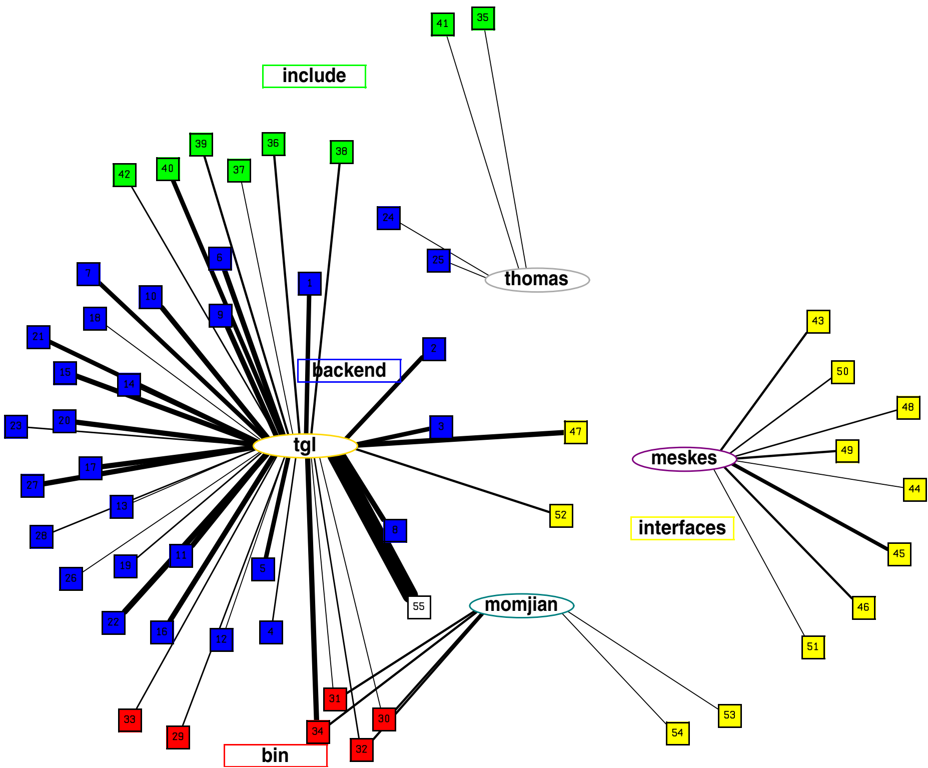

Fig. 16 Authorship graph during *Maintenance* period for PostgreSQL

interrelations for the purpose of predicting change or failures (Zimmermann et al., 2005; Ying et al., 2005; Hassan and Holt, 2004; Girba et al., 2004); these methods treat all codeMRs as equivalent. We speculate that using only fixDefectMRs and bugzillaMRs will improve the recall and precision of such methods (we plan to test this assertion in our future research).

The modification coupling and the authorship graphs provide useful and interesting visualizations of the files being modified and the authors of the modifications. In particular, the authorship graph can be used to discover “developer” interactions based on the assumption that if two developers work on the same file, they need to communicate (formally or informally). This graph also depicts the importance of modularization, since a well modularized product might reduce the required communication between developers.

One issue that must be addressed is how applicable to other projects are the observations made in this paper. Some of our observations appear to be general: the distribution of the number of files in MRs is very similar across different projects, regardless of their application domain; most MRs tend to contain very few files (the majority contain only one); in projects that use them, ChangeLogs are usually modified in all codeMRs, even if the change involves only one source code file. Another common observation is that commentMRs tend to be either very small (they modify one file) or very large (some modify hundreds and sometimes thousands of files).

There are, however, cases in which our observations are contradictory from one project to another. For example, we observed that in Evolution the smallest MRs are the bugzillaMRs, while in Apache these are the fixDefectMRs. If we were to use machine learning to automatically classify MRs, this might suggest that it is more important to use a training set from the project that is to be analyzed than using a generic training set derived from other projects.

One problem we faced in our classification was our inability to asses the amount of change in a given MR. Do different types of MRs change the source code in quantifiably different ways? For example, it might be possible that bugzillaMRs tend to modify very few functions, and they almost never change an API. Unfortunately this type of analysis requires more expensive, and language dependent, semantic analysis. The answer is left for future research. If we are able to understand better what different types of changes look like, we might be able to use this information for multiple purposes. For example, it could be used for visualization (the user can easily select different types of changes; for example he or she might be interested in architectural changes and not in defect fixes). Another advantage is that we could know which changes are more important for the analysis being performed (e.g., if we can detect which types of changes induce more bugs, we could better predict, for all new changes, which would be more likely to fail).

We have used the visualizations presented in this paper to understand how different open source projects have evolved, the development practices they have used, and how their developers are organized. However these visualizations still need to be evaluated by developers.

Finally, any analysis of the MRs of a project must be evaluated in the context of the development process that a project follows. This can be used to explain the disparities between two projects. For example, Apache MRs might be smaller because the project is very conservative and tends to make changes in very small, but well-tested increments. Also, some projects have more rigorous code submission policies. In Evolution, for example, any developer can commit any code without approval, while in Apache there is a formal procedure to determine what is added to the product (and what is not). This could explain why we detected a smaller proportion of bugzillaMRs in Apache than any other project.

5 Future Work

The historical logs of software projects are a wealth of data that can be used to further our understanding on how software changes. Similar studies like the one described here are needed, looking at different software products, in order to verify if some of the observations made herein extend to these projects. In particular, studies that involve other programming languages (such as Java) are needed.

Semantic analysis of MRs is needed to measure the amount of change performed. It is necessary to know what parts of the code change, and how (not only that the file has changed). In order to better classify MRs it is necessary to understand the "amount," and "type" of change that the MR performs. In other words, we need metrics that measure MRs.

The visualization of these data is also an interesting area of research. We need to improve our understanding of the needs of different types of users (developers,

managers and researchers) and then create visualizations that address their needs. And they should be evaluated to determine if they are useful or not.

One area we are particular interested in is the animation of the authorship and modification coupling graphs, with the intent of showing how the file's and author's interactions evolve through time.

6 Previous and Related Work

Mockus et al., (2002) analyzed Mozilla and Apache in an attempt to quantify aspects of developer participation and interaction, defect density and problem resolution intervals. Fisher and Gall have described a CVS fact extractor (Fischer et al., 2003b), and discussed the main challenges of creating a database of CVS historical data and then use this database to visualize the interrelationships between files in a project (Fisher and Gall, 2003). Fischer et al., (2003a) analyzed the modification records and describe different types of logical coupling among the files included in the MR. In German and Mockus (2003), we proposed methods to extract historical information (version control, defect tracking systems, ChangeLogs, mailings lists) to help in the empirical investigation of open source projects.

Several tools have been created to explore the historical data stored in version control repositories. Xia is a plugin for Eclipse that provides some visualization for CVS repositories (Wu et al., 2004b). Liu and Stroulia have developed JReflex, a plug-in for Eclipse for instructors of software engineering courses (Liu and Stroulia, 2003; Liu et al., 2004). It is designed to compare the differences in development styles in different teams, who does what, who works on what part of the project, etc. JReflex is intended to be a management oriented tool for browsing the CVS historical data.

Zimmermann and Weissgerber, (2004) described a method for the recreation of MRs from CVS logs similar to the one described in this paper (this paper was published at the same time as (German, 2004b) where we first described our algorithm). Their paper, however, does not include an analyses of the impact of varying the different parameters of the algorithm. In German (2004a,c), we demonstrated how version control logs can be used to understand how software is developed.

Historical information stored in version control systems has been used to assist in finding defects (Williams and Hollingsworth, 2005), estimated what files are more prone to defects (Ostrand and Weyuker, 2004; Ostrand et al., 2005; Graves et al., 2000), guide developers who have to complete tasks that are similar to one that was already completed (Zimmermann et al., 2005; Ying et al., 2005), predict what files are modified together (Hassan and Holt, 2004), or predict which files are to be modified in the near future (Girba et al., 2004). Hassan and Holt (2003) studied the complexity of software systems based on their source code history.

Few work has been done regarding the analysis of fine-grained changes as recorded by version control systems. Purushothaman and Perry, (2005) analyzed the changes made to a large proprietary system trying to understand what types of modification records (they called them modification requests) are more prone to introduce defects; the projects they analyzed included extensive documentation explaining the type of change, its rational and how it had been completed; unfortunately many projects do not record all this information.

With respect to visualizations, in Storey et al., (2005) we provided a comprehensive survey of visualization techniques used for historical information, including version control history. Of particular relevance to this research are the MDS-views that Fisher and Gall proposed (Fisher and Gall, 2003; Gall et al., 2003): they are graphs that show the dependencies between different parts of a system based on how frequently they were modified together. In MDS-views, nodes represent the different parts of the system (classes or files) and two nodes are connected if they are modified together in an MR that fixes a defect report. The distance between two nodes represents the "dissimilarity" between them. Like our research, they used the defect database (Bugzilla) in combination with the version control data. Collberg et al. proposed a system called GEVOL to explore the Evolution of a system using its CVS repository. Their graphs depict control flow, inheritance and control flow. In these graphs colour is used to distinguish the different author (Collberg et al., 2003). Eick et al., (2002) used historical information from proprietary systems to propose several visualizations; one of them, called "relationship between files" is very similar to our modification coupling graph. Their graph is not clustered by module and uses colour to depict frequency of modification (the modification coupling graph uses the thickness of the arc to depict this data, and colour to depict the module to which the file belongs).

Spectrographs (Wu et al., 2004a) and Evolution matrices (Lanza, 2001) are two dimensional grids that use historical information. One axis represents time and the other represents a part of the system (a class or a file, for example). If a part of the system is modified in a given time period, then the grid is marked accordingly. The basic idea of these visualizations is that they provide a quick overview of what parts of the system evolve at the same time.

7 Summary

In this study we developed an algorithm to reconstruct modification records (MRs) from CVS logs. We then analyzed the characteristics of these MRs, and proposed three classifications that can be done automatically: commentMRs (those MRs that only modify source code), bugzillaMRs (those MRs that correspond to an explicit defect fix as recorded by Bugzilla), and fixDefectMRs (MRs that include the word fix or bug and that seem to correspond to a defect fix). The characteristics of these subsets of MRs were compared across several projects. An important aspect of this study reported in this paper is our attempt to visualize the interrelations between authors and files as dictated by MRs. Two graphs were proposed: modification coupling of files (showing files that are modified together in the same MR during a given period) and authorship (showing, for a given period, authors and the files that they modified). We argued that these graphs can be used for several purposes; for example, to understand explicit and hidden relationships between files, to know who is working in a given part of the system, or what is the level of modularization in the project.

Acknowledgments This research was supported by the National Sciences and Engineering Research Council of Canada, and the Advanced Systems Institute of British Columbia. The author would like to thank the reviewers of this paper for their thoughtful comments that greatly improved the quality of this paper, and the Apache, Evolution, GNU gcc, Mozilla and PostgreSQL development teams.

References

Collberg C, Kobourov S, Nagra J, Pitts J, Wampler K (2003) A system for graph-based visualization of the Evolution of software. In: SoftVis '03: Proceedings of the 2003 ACM symposium on software visualization, ACM, New York, New York, pp 77-ff

Eick SG, Graves TL, Karr AF, Mockus A, Schuster P (2002) Visualizing software changes. IEEE Trans Softw Eng 28(4):396–412

Fischer M, Pinzger M, Gall H (2003a) Analyzing and relating bug report data for feature tracking. In: Proc. 10th working conference on reverse engineering, IEEE, pp 90–101

Fischer M, Pinzger M, Gall H (2003b) Populating a release history database from version control and bug tracking systems. In: Proceedings of the 19 IEEE international conference on software maintenance (ICSM'03), IEEE Computer Society, pp 23–32

Fisher M, Gall H (2003) MDS-views: visualizing problem report data of large scale software using multidimensional scaling. In: Proceedings of the international workshop on Evolution of large-scale industrial software applications (ELISA)

Gall H, Jazayeri M, Krajewski J (2003) CVS release history data for detecting logical couplings. In: Proceedings of the international workshop on principles of software Evolution (IWPSE), IEEE, pp 12–23

German DM (2004a) Decentralized open source global software development, the GNOME experience. Journal of Software Process: Improvement and Practice 8(4):201–215

German DM (2004b) Mining CVS repositories, the softChange experience. In: 1st international workshop on mining software repositories, pp 17–21

German DM (2004c) Using software trails to reconstruct the Evolution of software. Journal of Software Maintenance and Evolution: Research and Practice 16(6):367–384

German DM, Mockus A (2003) Automating the measurement of open source projects. In: Proceedings of the 3rd workshop on open source software engineering

German DM, Hindle A, Jordan N (2004) Visualizing the Evolution of software using softChange. In: Proceedings SEKE 2004 The 16th international conference on software engineering and knowledge engineering, Knowledge Systems Institute, 3420 Main St. Skokie, Illinois 60076, USA, pp 336–341

Girba T, Ducasse S, Lanza M (2004) Yesterday's weather: guiding early reverse engineering efforts by summarizing the Evolution of changes. In: Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04), pp 44–49

Graves TL, Karr AF, Siy H (2000) Visualizing software changes. IEEE Trans Softw Eng 26(7):653–661

Hassan AE, Holt RC (2003) The chaos of software development. In: Proceedings of the international workshop on principles of software Evolution (IWPSE), pp 84–95

Hassan AE, Holt RC (2004) Predicting change propagation in software systems. In: Proceedings of the 20th IEEE international conference on software maintenance (ICSM'04), pp 284–293

Lanza M (2001) The Evolution Matrix: recovering software Evolution using software visualization techniques. In: Proceedings of the 4th international workshop on principles of software Evolution (IWPSE), pp 37–42

Lerner J, Triole J (2000) The simple economics of open source. Working Paper 7600, National Bureau of Economic Research, http://papers.nber.org/papers/w7600

Liu Y, Stroulia E (2003) Reverse engineering the process of small novice software teams. In: Proc. 10th working conference on reverse engineering. IEEE, pp 102–112

Liu Y, Stroulia E, Wong K, German D (2004) Using CVS historical information to understanding how students develop software. In: 1st international workshop on mining software repositories, pp 32–36

Mockus A, Fielding RT, Herbsleb J (2002) Two case studies of open source software development: Apache and Mozilla. ACM Trans Softw Eng Methodol 11(3):1–38

Ostrand TJ, Weyuker EJ (2004) A tool for mining defect-tracking systems to predict fault-prone files. In: 1st international workshop on mining software repositories, pp 85–89

Ostrand TJ, Weyuker EJ, Bell R (2005) Predicting the location and number of faults in large software systems. IEEE Trans Softw Eng 340–355

Purushothaman R, Perry DE (2005) Toward understanding the rhetoric of small source code changes. IEEE Trans Softw Eng 31(6):511–526

Storey MA, Čubranić D, German DM (2005) On the use of visualization to support awareness of human activities in software development: a survey and a framework. In: Proceedings of the 2nd ACM symposium on software visualization, pp 193–202. To be presented

Williams CC, Hollingsworth JK (2005) Automatic mining of source code repositories to improve bug finding techniques. IEEE Trans Softw Eng 31(6):466–480

Wu J, Holt RC, Hassan AE (2004a) Exploring software Evolution using spectrographs. In: Proc. 11th working conference on reverse engineering, pp 80–89

Wu X, Murray A, Storey M-A, Lintern R (2004b) A reverse engineering approach to support software maintenance: version control knowledge extraction. In: Proc. 11th working conference on reverse engineering, pp 90–99

Ying A, Murphy GC, Ng R, Chu-Carroll MC (2005) Predicting source code changes by mining change history. IEEE Trans Softw Eng 31(9):574–586

Zimmermann T, Weissgerber P (2004) Preprocessing CVS data for fine-grained analysis. In: 1st international workshop on mining software repositories, pp 2–6

Zimmermann T, Weissgerber P, Diehl S, Zeller A (2005) Mining version histories to guide software changes. IEEE Trans Softw Eng 31(6):429–445

Daniel M. German is assistant professor in the Department of Computer Science at the University of Victoria. His main areas of research are software evolution, open source software development and intellectual property.