International Journal of Software Engineering
and Knowledge Engineering
Vol. 16, No. 1 (2006) 5–21
© World Scientific Publishing Company

VISUALIZING THE EVOLUTION OF SOFTWARE USING SOFTCHANGE

DANIEL M. GERMAN* and ABRAM HINDLE†

Software Engineering Group, Department of Computer Science,
University of Victoria, Victoria, BC, Canada V8W3P6
* *dmgerman@uvic.ca*
† *abez@abez.ca*

A typical software development team leaves behind a large amount of information. This information takes different forms, such as mail messages, software releases, version control logs, defect reports, etc. softChange is a tool that retrieves this information, analyses and enhances it by finding new relationships amongst it, and then allows users to navigate and visualize this information. The main objective of softChange it to help programmers, their management and software evolution researchers in understanding how a software product has evolved since its conception.

Keywords: Software evolution; software trails; CVS; visualization; softChange.

1. Introduction

Many software projects use a version control repository to record the evolution of their source code. These repositories keep track of every change to any source file of the project, including metadata about the change, such as author and date when it happened. Over time, the amount of revisions to a project become enormous. For example, the Mozilla project is composed of 35,000 files which have been modified 450,000 times in 5.5 years of development (from March 1998 to August 2003) by 500 different developers.

CVS, the Concurrent Versions System, is arguably the most widely used version control management system available in the market and has become a de-facto standard in the development of open source projects.

While CVS is a very powerful tool, it provides many barriers to the extraction and visualization of valuable information. CVS commands are cryptic and their output formats are not easy to understand. CVS queries often produce an excess of information which is hard for the frustrated developer to sift through. General summaries are rarely provided. Furthermore, CVS does not provide any alternative methods to browse through its information.

CVS is built around a group of command-line programs. Several GUI applications (winCVS, tkCVS, cvsWeb, LinCVS, Pharmacy, gCVS, etc.) and some integrated development environments (such as Eclipse) provide a GUI to CVS. In all these cases, the tools are created around the CVS commands and options, providing nothing more than a fancy GUI to the actual commands.

One of the main disadvantages of CVS is that it is not transaction oriented. In other words, when a developer proceeds to "commit" a group of changes to a number of files, CVS does not keep track of all the files modified by this commit operation. It treats each change to a file independently of the other files included in the commit. After the commit has taken place, CVS does not know which files were modified together — users can take steps to make their commits more traceable by using the same log comment, or tagging the files in their commit. This information, however, is important because it highlights coupling amongst files: if two files are modified at the same time, it is because they share *something* in common (at the very least those two files were checked-in together). In this paper we refer to a commit operation as a *modification record* (MR). An MR is therefore a collection of revisions to files that are modified at the same time. Several heuristics have been proposed to recover these MRs.

The information stored in the CVS repository is quite valuable as it can help answer many questions. For instance, it can assist developers to find who has modified which files and when; it can also help the administration to understand the modification patterns of the project and the way the different team members interact. Finally, it can help to recover the evolution of the project. In [14] Wu collected questions that developers might ask of a CVS repository: what happened since I last worked on this project? Who made this happen? When did the change take place? Where did the change happen? Why were these changes made? How has the file changed? What methods or functions were changed? What is the frequency of change? What files have changed? Who is working on each module?

Administrators, on the other hand, are interested in higher level questions and metrics such as: how often does a programmer complete an MR? How much does the programmer modify in an MR? What kind of commits does one programmer do? How much has changed between each release? How many bugs are fixed and found after a stable release? What kind of modifications are done at a certain time? When was a module stabilized? What is the daily LOC count for each programmer? When is a module being developed and maintained?

We define *software trails* as information left behind by the contributors to the development process, such as mailing lists, websites, version control logs, software releases, documentation, and the source code [8]. In this paper we describe softChange, a tool that mines software trails from CVS and then enhances this data with some heuristics in order to recover higher level information, such as rebuilding MRs. Each MR is analyzed in order to know what type of changes took place; such as adding new functions, reorganizing source code, adding comments to the code only, etc. After extraction and analysis, softChange provides a graphical and hypertext representation of this information.

This paper is divided as follows: previous work is described in Sec. 2; Sec. 3 describes softChange; Sec. 4 describes the visualization features of softChange; in Sec. 5 we demonstrate the effectiveness of softChange by analyzing two mature projects; we end with our conclusions, and future work.

2. Previous Work

The two most commonly used hypertext front ends to CVS are Bonsai [10] and lrx [9]. They provide a Web interface to the CVS repository and isolate the user from the complexities of the CVS commands (the man page of CVS is 9000 words long). Both tools allow the user to inspect the history of any given file in the project and neither of them attempts to enhance the software trails available in the repository.

Xia is a plugin for Eclipse for the visualization of CVS repositories [14]. Xia recovers relations available in the logs of a CVS repository and allows the user to navigate them. It uses squares to represent files, their revisions and developers, and lines to represent the relationships between them. Xia has two main limitations. The first is that Xia relies on the Eclipse API to access the CVS repository. Every time Xia wants to create a view, it queries the CVS repository in order to retrieve the necessary data. This becomes a very expensive operation making Xia extremely slow in large CVS repositories. The second limitation is that Xia operates at the revision level, not at the MR level.

Hipikat aggregates many sources of information such as Bugzilla, the CVS repository, mailing lists, emails etc. and provides a searchable query interface [1]. The purpose of Hipikat is to "recommend software artifacts" rather than summarize and visualize them. Thus Hipikat is much like Google for a software project. One interesting feature of Hipikat is that it correlates software trails from different sources, inferring relationships between them. Liu and Stroulia have developed JReflex, a plug-in for Eclipse for instructors of software engineering courses. JReflex helps the instructor to monitor how different teams of students developed a term project by using their CVS historical information [11]. 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. JReflex does not enhance the information available in CVS. Fisher and Gall have described a CVS fact extractor in [3]. In it they describe the main challenges of creating a database of CVS historical data and then use it to visualize the interrelationships between files in a project [4]. Spectrographs are a type of visualization based on historical data that attempts to show where and where change occurs in a system [13]. MDS-Views are another type of visualization that shows dependencies between different parts of the system based on how frequently they are modified to solve a given defect report. It is intended to be used by managers and researchers who are interested in evaluating the current "design erosion" of the system [2]. In [12] we propose a framework for describing, comparing and understanding visualization tools that provide awareness of human activities in software development, including tools that mine and use historical information to achieve their goal.

3. softChange

Early in our research we gathered a number of requirements for a tool to explore software trails:

- The main users of this tool will be researchers (ourselves) who are interested in understanding how software evolves.
- The software trails should be extracted and stored in a database for further analysis. Ideally the tool should retrieve each trail only once.
- The tool should have a layered architecture: one layer dedicated to the extraction of the trails, and another dedicated to the exploration of this data. A third layer should be responsible for the analysis and discovery of new facts from the ones currently stored in the database. The database should serve as the intermediary between these layers. These layers should be as independent as possible from each other.
- The database schema should be extensible, that is, as more software trails are extracted (and new relationships are found) these can be incorporated into the database.
- The tool should provide a way to navigate the information at different levels of granularity: sometimes one is interested in the specifics of a given change, while in some other cases in trends.

We proceeded to create softChange around this set of requirements.

3.1. softChange *architecture*

softChange is composed of four main components, depicted in Fig. 1.

- Software trails repository: At the core of softChange lies a relational database that is used to store all the historical information (see [7] for a description of the schema of this database).
- Software trails extractor: In a typical software development project, software trails originate from many different sources: CVS historical data, email messages, bug reports, ChangeLogs, etc. The purpose of softChange trails extractor is to retrieve as many software trails as possible. Currently, softChange is able to retrieve trails from CVS, from ChangeLogs, from the releases of the software (the tar files distributed by the software team) and from Bugzilla.
- Software trails analyzer: Once softChange has extracted the software trails, it proceeds to use this information to generate new facts. For example, using a set of heuristics, softChange regroups file revisions into MRs [7]. softChange analyzes the changes in the source code and thus extracts a list of function, methods and classes that have been added, modified or removed from one file revision

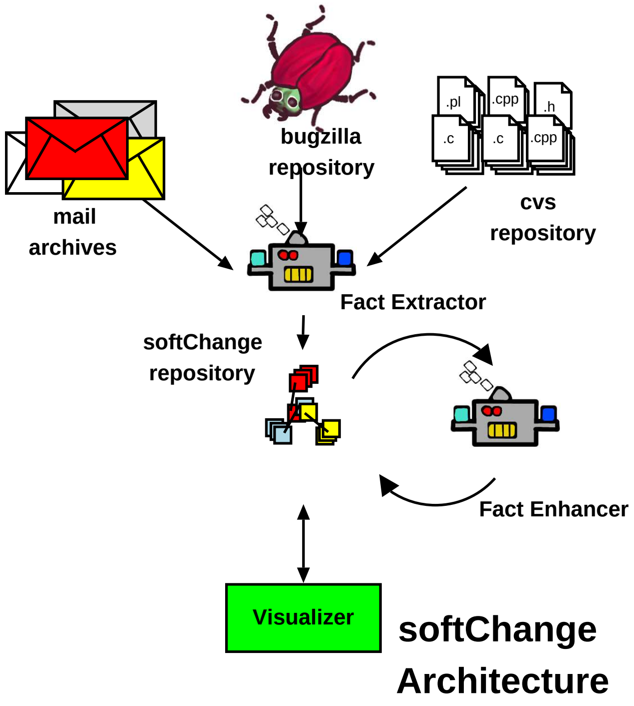


Fig. 1. Architecture of softchange.

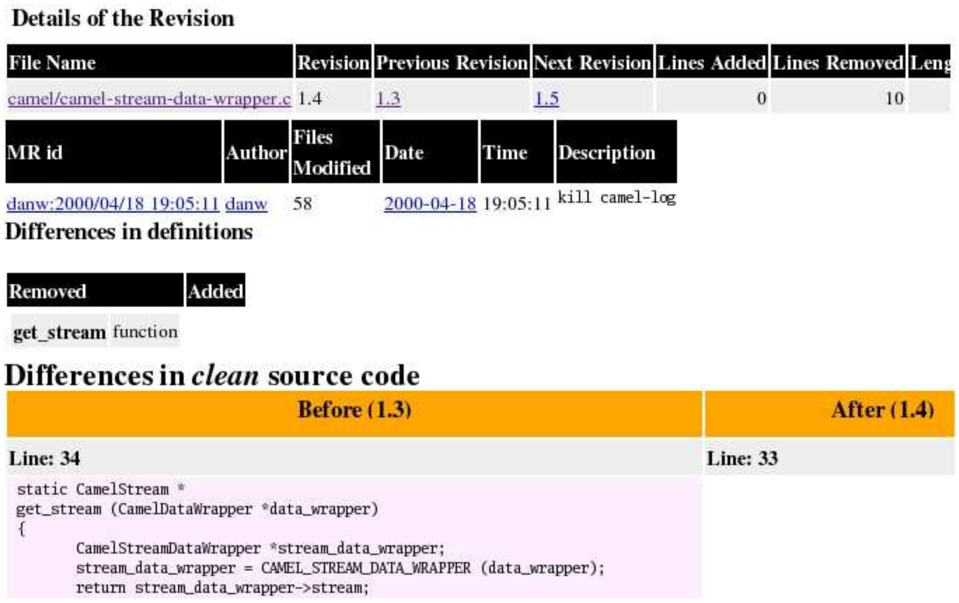
to the next. softChange also correlates the available software trails; for example, softChange links a given MR to its Bugzilla bug report.

- Visualizer: softChange provides a visualizer to the repository that allows the user to explore the software trails. This front end is described in detail in the next section.

4. Visualizing Software Trails

The main objectives of softChange are to summarize, browse and visualize the evolution of a software project. The visualizer of softChange is divided into three main components: a browser, a chart generator and an interactive graphical visualizer.

The softChange browser is a hypertext application that permits the exploration of MRs, the files they contain, and any changes made to these files. These changes include: a list of the functions added, modified or deleted, a pretty-printed version of the differences between source code, both in its original form, and after the comments and empty lines have been removed (which we call *clean* version). The advantage of generating a *clean* version of the modified code is that it allow us to detect changes that are only in whitespace and comments and therefore do not change the functionality of the source code. For example, in the history of PostgreSQL (the



Details of the Revision

File Name	Revision	Previous Revision	Next Revision	Lines Added	Lines Removed	Leng
camel/camel-stream-data-wrapper.c	1.4	1.3	1.5	0	10	

MR id	Author	Files Modified	Date	Time	Description
danw:2000/04/18 19:05:11	danw	58	2000-04-18	19:05:11	kill camel-log

Differences in definitions

Removed	Added
get_stream function	

Differences in *clean* source code

Before (1.3)	After (1.4)
Line: 34	Line: 33

```
static CamelStream *
get_stream (CamelDataWrapper *data_wrapper)
{
        CamelStreamDataWrapper *stream_data_wrapper;
        stream_data_wrapper = CAMEL_STREAM_DATA_WRAPPER (data_wrapper);
        return stream_data_wrapper->stream;
```

Fig. 2. A screenshot of the browser showing the details of a revision.

free software database management system) we detected that approximately 20% of all file revisions are only changes in comments or whitespace; for Evolution (a free software mail client similar to Microsoft Outlook) the proportion was approximately 9%. The softChange browser targets users who are interested in explicit changes, such as what are the changes that occur in this file in a given date; or for a given change, which methods were modified, or what other files were modified at the same time.

The main goal of the softChange browser is to help users in the exploration and understanding of the history of a given project. A developer can quickly navigate through the MRs, revisions or defects of a project. Figure 2 shows the details of a revision from the project Evolution. It contains an explanation of the change, what file the revision corresponds to, and which MR the revision is part of. In this example, the softChange browser also indicates that one function has been deleted and shows the corresponding code that was removed. The user can easily navigate between the different revisions of the same file, to other revisions that compose the MR, or to other changes made by the same author.

The second visualizer of softChange is a chart generator that plots different types of information as two-dimensional graphs. Some of the charts created by softChange are:

- Growth of LOCS vs. time: Although this metric is not the most valuable metric, it does give an indication of activity and the possible kinds of activities occurring in the project. A rapid rise could indicate development of new functionality whereas a constant slow rise could indicate maintenance activity or lack of activity.

- Number of MRs vs. time: This metric seems to better represent the volume of activity in a repository. This metric is useful to observe change at the maintenance level whereas LOCS vs. time did not provide much information about the actual activity of a repository.
- Number of files vs. time: This metric usually indicates structural changes to a repository whether it is adding or removing modules, refactoring parts of modules or just renaming files. Since the adding of files is much more rare than modifying files, this metric indicates behavior which could be of significance.
- Number of files per MR: This metric is useful to study the type of changes that go into a system. We have observed that defect fixes involve a small number of changes, and that changes in comments tend to include a large number of files.
- Proportion of MRs per contributor: MRs can be seen as an indicator of finished tasks, and therefore this plot can be used as an indicator of the amount of work by a developer.
- Frequency of modification to modules: This measurement could indicate which modules undergo a lot of development and which ones are relatively stable.

We are constantly experimenting with new charts that might highlight certain aspect of the history of the project. The objective of these charts is to give the user a high-level overview of how the project is evolving. The user is allowed to select the period of interest and the output is a Postscript file.

The final component of softChange is its graphical visualizer. It is an interactive application that allows the user to explore the history of the project as a collection of graphs. softChange's graphical visualizer creates many graphs:

1. A *file authorship* graph relates files and the contributors who modify them. Each file and each contributor is represented by a node. A file is connected to the contributors who have modified it. The arc's width is proportional to the number of times a contributor has modified the corresponding file.
2. A *coupling graph* shows which files are modified together. Each file is a node, and two files are connected by an arc if they are modified together. The arc's width is proportional to the number of times the two files have been modified together.
3. A *change overview* graph shows all files in a project and highlights those that have been modified. Each node is a file, and they are connected in a tree according to their organization in the file system.
4. An *authorship overview* graph shows who is the most frequent person to modify a given file.
5. A *file evolution* graph shows when a file is modified and by who. This graph is similar to the spectrograph proposed in [13].

5. Evaluation

We have been using softChange as part of our research in software evolution and software engineering empirical studies. In [8] we described a method to recover the evolution of a software system using its historical information (version control, releases, email, change logs, and defect information). We used softChange to extract the history of Evolution (a mail client for Unix), and then analyze and visualize it. softChange was instrumental in allowing us to look into the past of the project and infer how it had evolved since its conception.

In [5] softChange was used to extract historical information from the free software project GNOME (a large project with more than 500 developers who are working towards building a suite of libraries and GUI applications for the Linux desktop). In this project we were interested in understanding the way that the software developers of the GNOME project collaborated. The analysis of its history allowed the discovery of interesting facts about the project: its growth, the interaction between its contributors, the frequency and size of the contributions, and the important milestones in its development.

In [6] we looked in detail into the characteristics of MRs in the Evolution project: the interrelationships of the files that compose them, and their authors, the type of work that each MR completed, etc. We proposed several metrics to quantify MRs.

We proceed to exemplify the use of softChange to compare and visualize the history of two software projects: PostgreSQL and Evolution.

5.1. *Methodology*

The Evolution project was born in 1999. In November 2003 we proceeded to extract its software trails. Its CVS repository included 5127 files, which had been modified 47,814 times, by 148 contributors. We correlated this information with its Bugzilla repository. We also retrieved every release (a tar file available for download by any user) available.

In September 2004 we proceeded to extract all the available software trails from the PostgreSQL CVS repository. Unfortunately its CVS repository only records its history starting from July 1996 (version 1.02). Its developers did not use CVS prior to this date. PostgreSQL does not use Bugzilla for its defect management. Its CVS repository included 5581 files which were modified 91740 times by 27 different contributors.

We then proceeded to create various charts and graphs in order to show how they can be used to assist *software evolutionists* to recover the history of these projects. It is important to mention that the objective of this section is not to compare these two projects nor the software practices of its developers. Both projects are very different in nature: Evolution has a very rich graphical user interface, while PostgreSQL is a powerful DBMS, and therefore they are expected to show strong differences. Our intention, instead, is to show that softChange can be used to highlight these differences. It is up to the *software evolutionist* to interpret these differences.

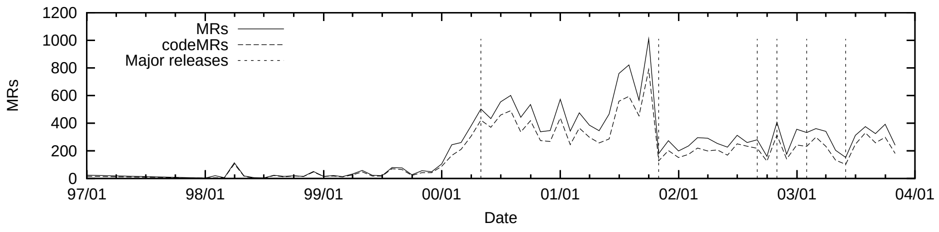


Fig. 3. Number of MRs per month for Evolution.

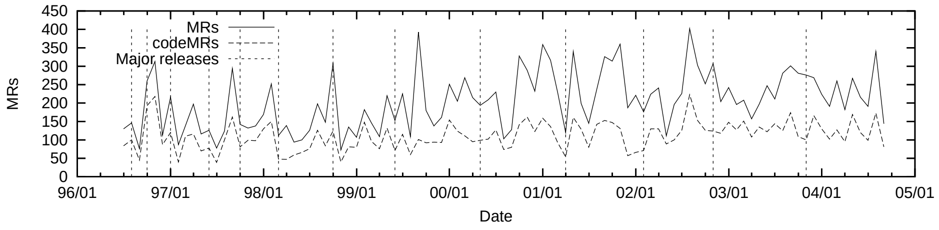


Fig. 4. Number of MRs per month for PostgreSQL.

5.2. *Visualizing evolution using* softChange

Understanding how a software system evolves requires a comprehensive analysis of all available historical information. One of the main challenges a researcher faces is information overloading: there is too much information to inspect. Visualizations can be used to overcome this problem. Several visualizations for software trails have been proposed in the literature [12] and each is intended to provide a different view of the system with a particular purpose in mind.

We will proceed to show a sample of the visualizations created with softChange. We start with a comparison of MRs over time. The intention of this plot is to show a measure of activity in the project. Figures 3 and 4 correspond to the number of MRs per month for Evolution and PostgreSQL. These charts highlight a special type of MR that is of particular interest: MRs that contain at least one source code file. We call these codeMRs. Note that the proportion of codeMRs to MRs remains fairly constant over the life of both projects. There seem to be less codeMRs per MR in PostgreSQL, however. The chart for the project Evolution shows how the project evolved slowly during its beginning, while PostgreSQL has remained relatively steady during the last 8 years. It is also interesting that there is a slight periodicity in these graphs that matches the periods between releases.

In order to create the *file authorship* and the *coupling graph* we are interested in finding MRs that contain files that show a significant level of "coupling" between themselves. We have discovered that some MRs are only modifications to comments (for example, the largest MR in PostgreSQL contained 995 files and it was a simple modification to the comment of each of these files). We proceeded to filter codeMRs

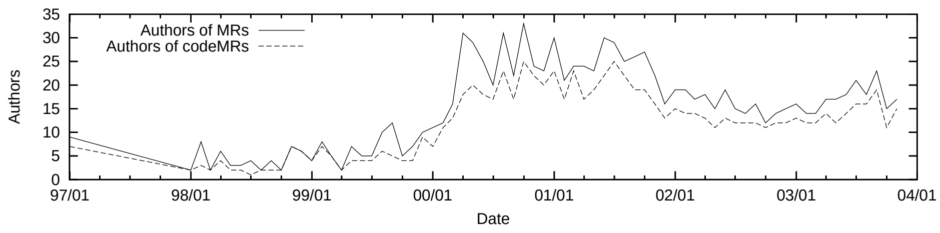


Fig. 5. Number of authors per month for Evolution.

using the following algorithm:

- Eliminate codeMRs that are only modifications of comment. These types of modifications do not necessarily imply a functionality relationship between the files that are included (they might be related from other points of view, though). For example, the change in the license or the copyright of a set of files.
- Eliminate codeMRs that are not committed to the main trunk of the repository. Branches in CVS pose difficult problems for researchers because the analysis can potentially record changes twice: one when the MR is committed to the branch, and when the branch is merged back to the main trunk. Branches can also be experimental work that is never added to the project [3].
- Eliminate codeMRs that contain more than 25 source files. In our observations large codeMRs do not provide very useful coupling information because they are few (in the projects we have analyzed they account for less than 3% of the MRs) and they might contain files that are not really related (for example branch merges tend to be large, or an MR might implement different features at once).

Our assumption is that the remaining codeMRs will contain files that are more likely to be related to each other than if we used all of the original MRs. We chose to extract codeMRs for only one month of development. We chose October 2002 for Evolution, and January 2002 for PostgreSQL as these dates corresponded to the month just before a major release (version 1.2.0 of Evolution was released on 2002/11/07, and version 7.1 of PostgreSQL was released 2002/02/04). Our analysis of the comments of the MRs during these periods allow us to state that they were periods of debugging and maintenance and very few features were added to the systems.

The file authorship graph is intended to show how many contributors tend to modify a given file. These graphs can be very useful to explore the interrelationships between developers and to show how independently they work. It can also tell who is responsible for a given file. One can argue that a system that has been properly divided among its programmers, should have an authorship graph that contains clusters of nodes that are slightly interconnected between each other (most files are modified by one individual — the clusters, and some are modified by several individuals). Figures 6 and 7 show the authorship graph for both projects.

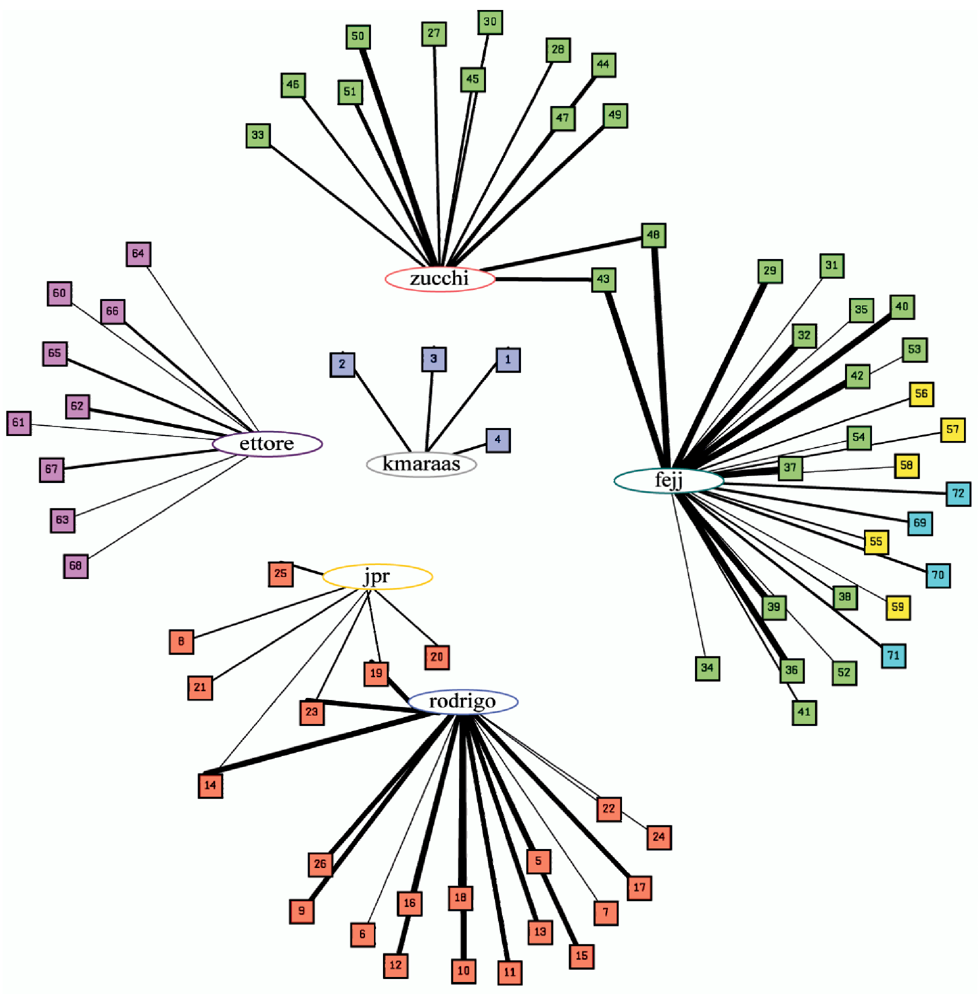


Fig. 6. Authorship graph for the project Evolution for November 2002.

The coupling graph can show hidden relationships between files that are not apparent from static analysis. For example, a program might have a function that writes to a file and another one that reads it, but they might not call each other and static analysis will not relate them. They might, however, need to be modified together if there is a change in the format of that file. The coupling graph for PostgreSQL is shown in Fig. 8 during the month of January 2002. The graph shows that most files create small clusters and that some files connect sub-clusters.

The files activity graph is intended to show how localized the changes to files are during a given period. Figure 9 shows the file activity graph for PostgreSQL during January 2002. The darker a file is, the more frequently it has been modified. The graph shows the files in the `backend` module of the database. The authorship

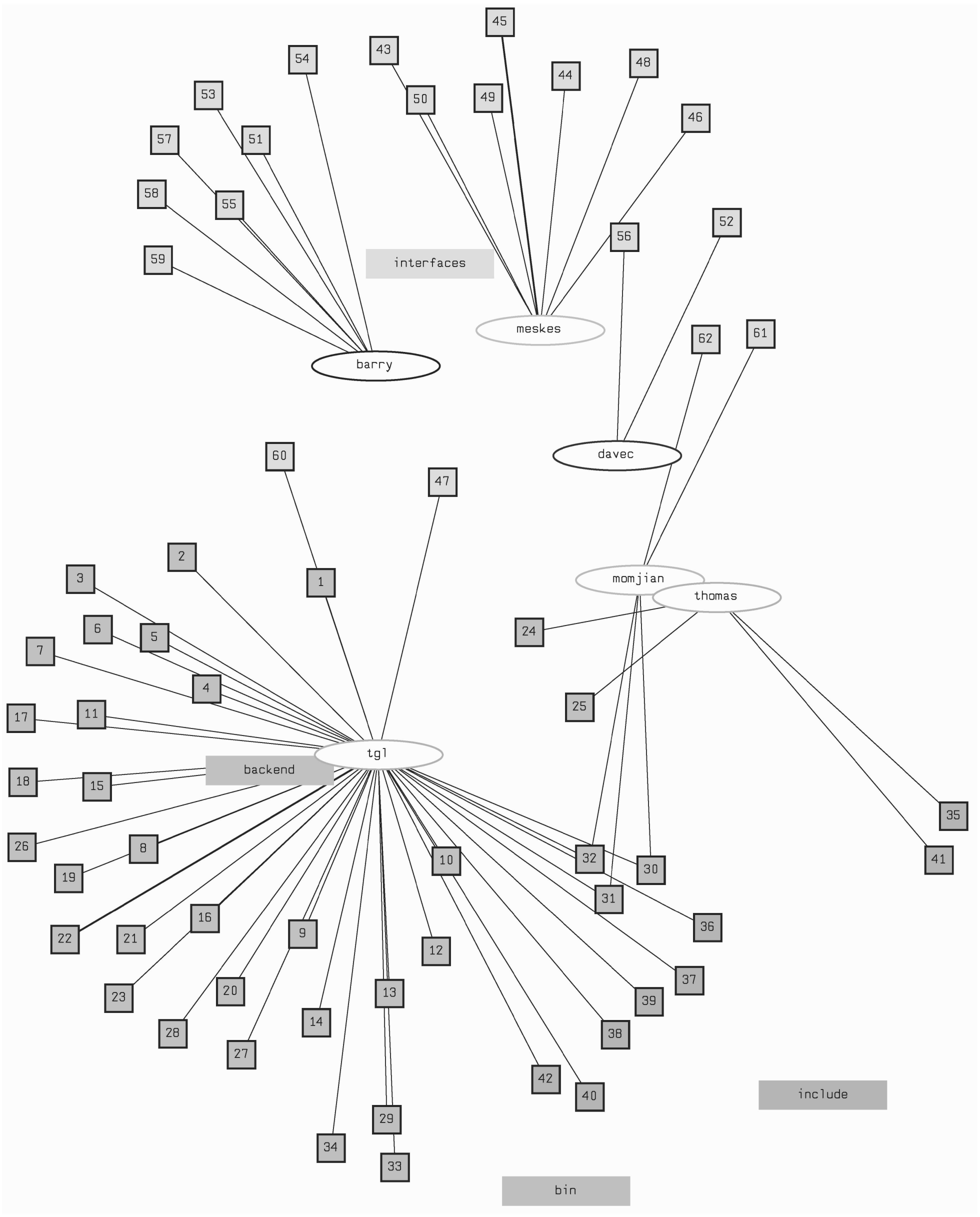


Fig. 7. Authorship graph for the project PostgreSQL for January 2002.

overview graph is similar to the files activity graph but it only shows files modified during the given period, and it colors the files according to the author who modified them the most. Figure 10 shows the corresponding graph for the same module and the same period.

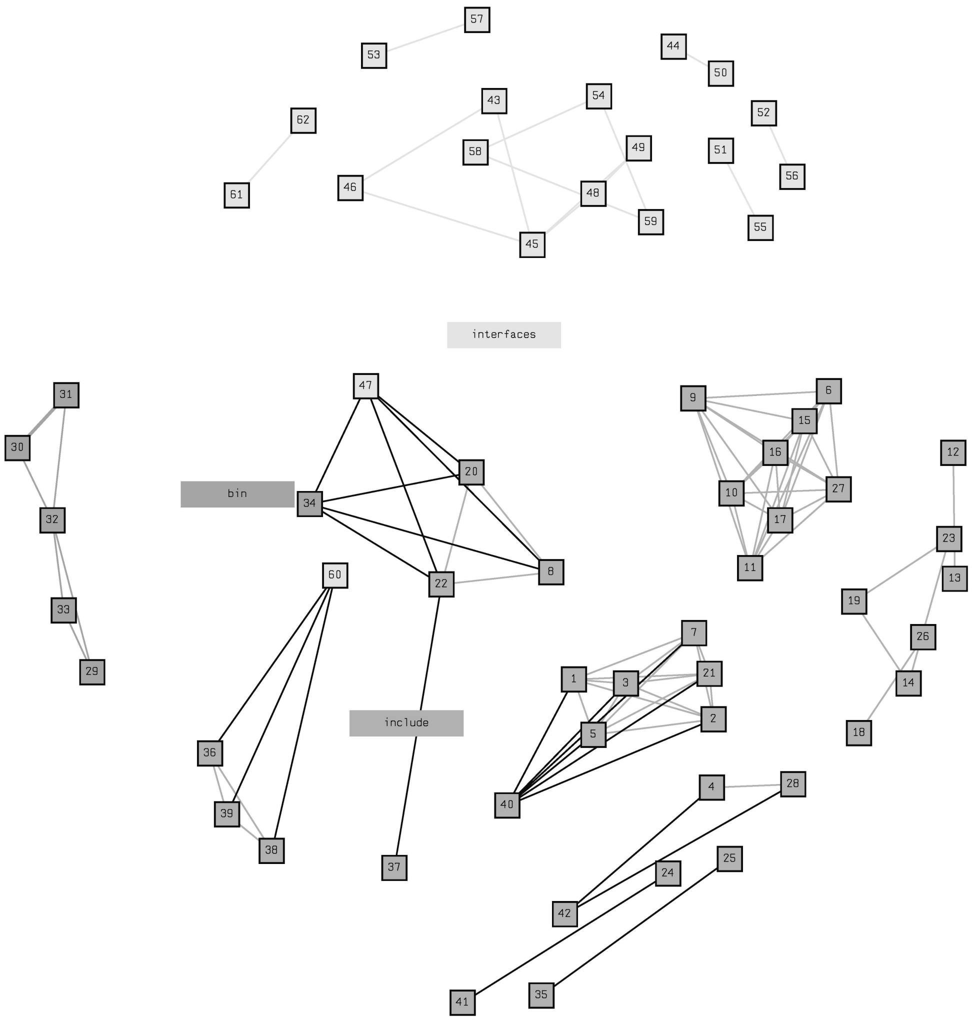


Fig. 8. Coupling graph for the project PostgreSQL for January 2002.

The file evolution graph tracks the modifications of files through time and can also be used as an indicator of where the system is being modified and it can also highlight files that are changed during the same periods. Figure 11 shows a section of an evolution graph for Evolution. It is a matrix view where rows depict files and columns correspond to periods of time (their length is selectable by the user). If a file was modified during a particular period, then the intersection is colored according to which contributor has modified the file most frequently during that period. The goal of the graph is to show an overview of the history of the project, where change occurs during any given period and who is responsible for it. In this example one can observe that some files are modified almost daily (in this

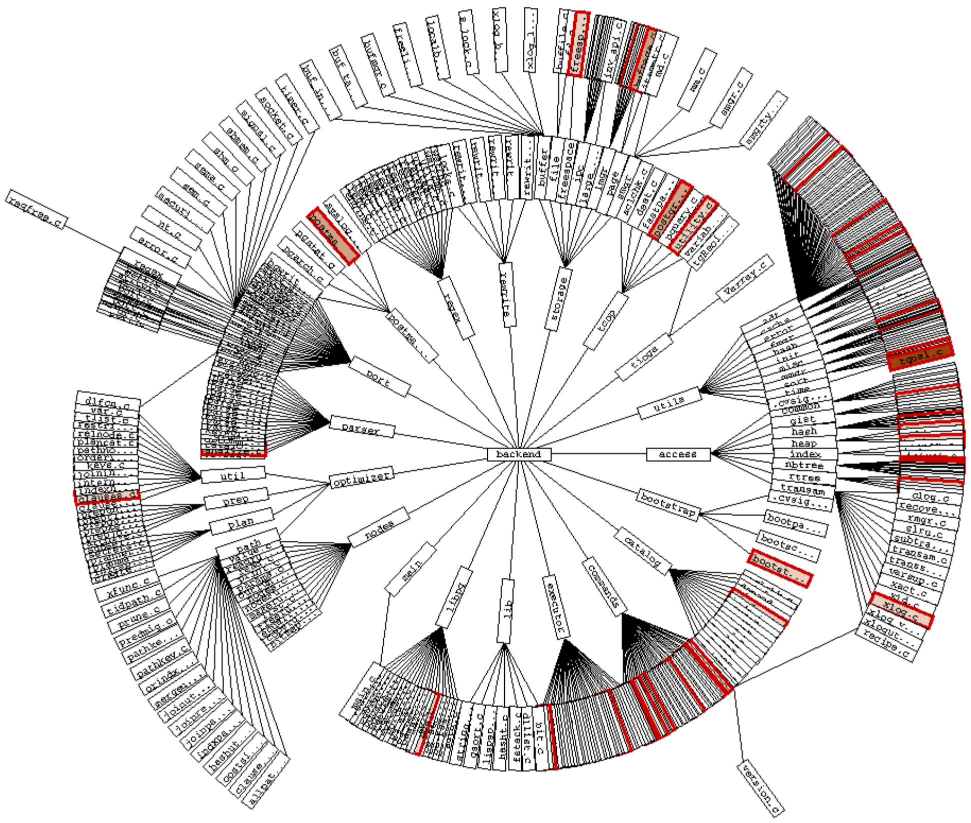


Fig. 9. File activity graph for the project PostgreSQL for January 2002.

example it was the ChangeLog, a file that documents changes and that developers are expected to update every time they make changes), and that some days multiple files are changed (an example of a change that spans several files is a change in the license or the copyright of the project, which usually involves changing every source file in the project).

softChange has only been used by our research group. We have used it successfully to analyze and understand the evolution of software. Because we have created it based on the questions in Sec. 1 and in response to our own needs, we are confident that it will be useful to other users too.

It is also important to mention that, in its current state, softChange is not a tool that gives answers on how software evolves. Instead, it is a tool to assist the software evolutionist in the exploration of the history of a software project. The evolutionist needs to apply her experience and insight to explain how the software has evolved. softChange provides tools to do that exploration but it does not answer the question "how has this project evolved". It is up to the *evolutionist* to answer that question.

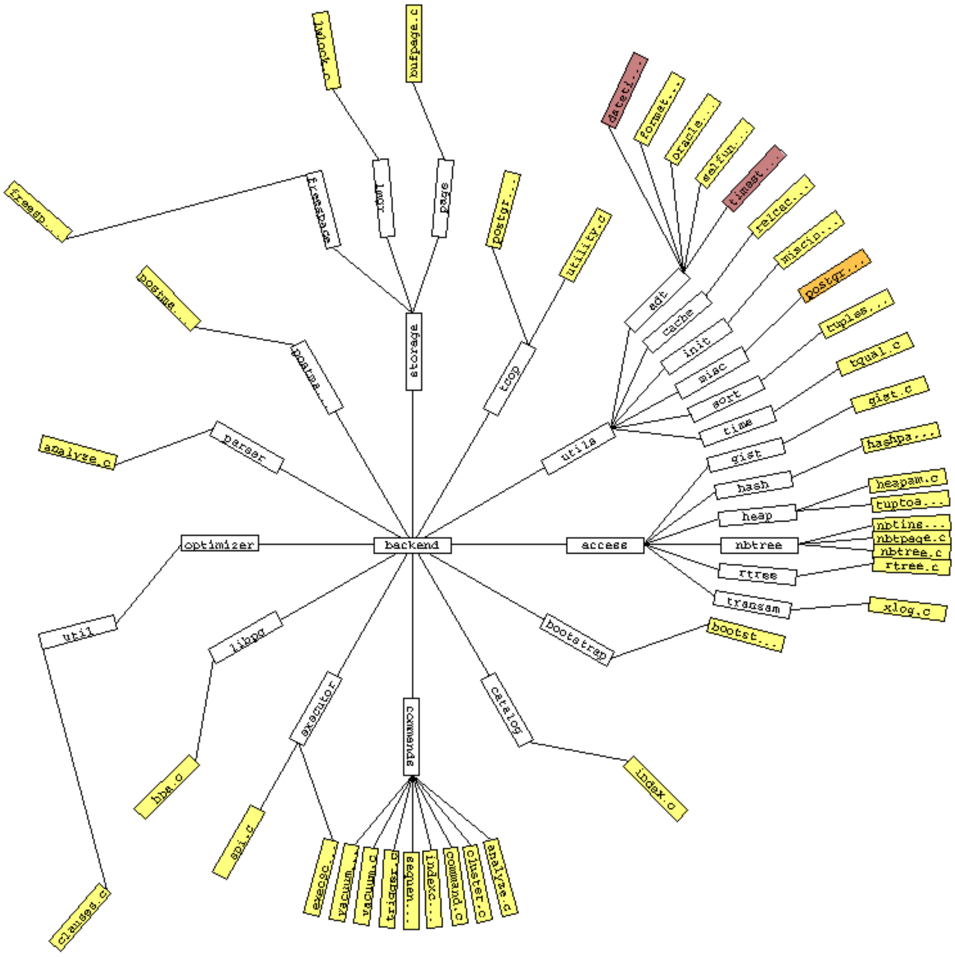


Fig. 10. Authorship overview graph for the project PostgreSQL for January 2002.

6. Future Work and Conclusions

One of the main advantages of keeping all software trails in a relational database is that we can analyze and enhance them by extracting new knowledge from them. Our current research investigates the characterization of MRs. We want to know what types of MRs are typically committed by developers: are they source code modifications, documentation, or internationalization? If there are changes to the source code, are they bug fixes, new features, reorganization of the code or clean up? With this enhanced information, users can discriminate and select the changes they are interested in, without being overwhelmed by the amount of available data. The more facts that are known about the evolution of the project, the better the visualization tools that can be created.

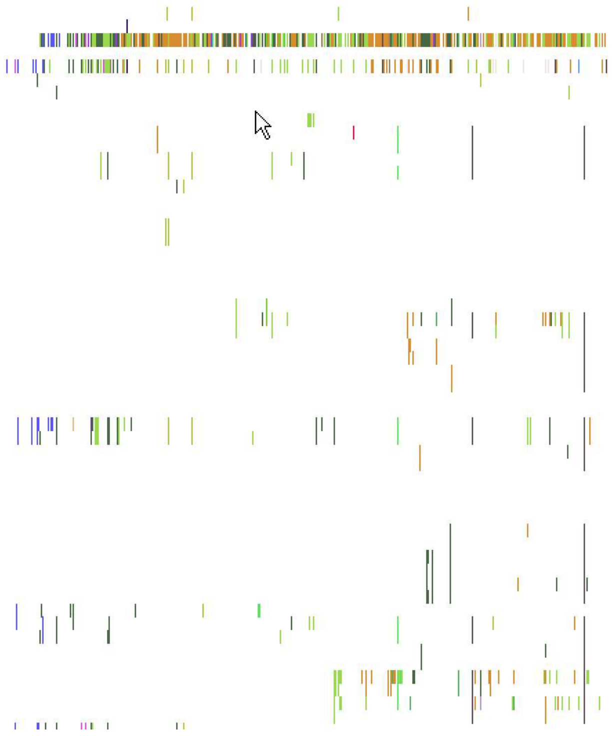


Fig. 11. Detail of the evolution graph for the project Evolution.

We are interested in formally evaluating the usefulness of softChange. So far we have been its only users and its development has been driven by our instincts and the needs of our research. It will be valuable to ask developers and other interested parties (such as managers and other researchers) what are their needs and how they believe they can be satisfied.

Data mining of software trails is a promising area of research. There might exist many hidden relationships in the data extracted that could be exploited: for example, do some developers tend to create better or worse code? Are some files more prone to defects than others? Do some code constructs create defect-prone code? Old, stable software projects have a large amount of software trails available. These trails can be mined for new facts; these facts can be used to produce better visualizations. softChange is open source and it is available under the General Public License. We are currently looking for software developers, and researchers interested in using it.

We have presented softChange, a system for the visualization of the history of the development. softChange retrieves and analyses the available software trails from a CVS repository and stores them into a relational database. softChange is composed of a hypertext application and a visualization tool that creates several graphs and charts. softChange is intended for three main groups of users: developers, their management and researchers. We exemplified the use of softChange with two different mature software projects.

Acknowledgments

This research was supported by the National Sciences and Engineering Research Council of Canada, and the Advanced Systems Institute of British Columbia. We thank Norman Jordan, Blake Dickie, Tao Xia, Greg Hall and Andrew Swerdlow for their contributions to softChange. We also want to thank the reviewers of this paper for their valuable comments.

References

1. D. Cubranic and G. C. Murphy, Hipikat: Recommending pertinent software development artifacts, in *Proc. 2003 Int. Conf. on Software Engineering*, Association for Computing Machinery, Portland, May 2003, pp. 408–418.
2. M. Fischer and H. Gall, MDS-Views: Visualizing problem report data of large scale software using multidimensional scaling, in *International Workshop on Evolution of Large Scale Industrial Software Applications* (*ELISA*), 2003.
3. M. Fischer, M. Pinzger, and H. Gall, Populating a release history database from version control and bug tracking systems, in *Proc. Int. Conf. on Software Maintenance*, IEEE Computer Society Press, September 2003, pp. 23–32.
4. M. Fisher and H. Gall, MDS-Views: Visualizing problem report data of large scale software using multidimensional scaling, in *Proc. Int. Workshop on Evolution of Large-Scale Industrial Software Applications* (*ELISA*), September 2003.
5. D. M. German, Decentralized open source global software development, the GNOME experience, *J. Software Process: Improvement and Practice* **8**(4) (2004) 201–215.
6. D. M. German, An empirical study of fine-grained software modifications, in *20th IEEE Int. Conf. on Software Maintenance* (*ICSM'04*), September 2004, pp. 316–325.
7. D. M. German, Mining CVS repositories, the softChange experience, in *1st Int. Workshop on Mining Software Repositories*, May 2004, pp. 17–21.
8. D. M. German, Using software trails to reconstruct the evolution of software, *J. Software Maintenance and Evolution: Research and Practice* **16**(6) (2004) 367–384.
9. A. G. Gleditsch and P. K. Gjermshus, lrx Cross-Referencing Linux, http://lxr.sourceforge.net/, visited Feb. 2004.
10. T. Hernandez, The Bonsai Project, http://www.mozilla.org/projects/bonsai/, visited Feb. 2004.
11. Y. Liu and E. Stroulia, Reverse engineering the process of small novice software teams, in *Proc. 10th Working Conf. on Reverse Engineering*, IEEE Press, November 2003, pp. 102–112.
12. M. A. Storey, D. D. Čubranić, and D. M. German, On the use of visualization to support awareness of human activities in software development: A survey and a framework, in *Proc. 2nd ACM Symposium on Software Visualization*, 2005.
13. J. Wu, R. C. Holt, and A. E. Hassan, Exploring software evolution using spectrographs, in *Proc. 11th Working Conference on Reverse Engineering*, 2004, pp. 80–89.
14. X. Wu, Visualization of version control information, Master's thesis, University of Victoria, 2003.