JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2004; **16**:367–384
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.301

Research

Using software trails to reconstruct the evolution of software

Daniel M. German*,†

Department of Computer Science, University of Victoria, Victoria, BC, Canada V8W 3P6

SUMMARY

This paper describes a method to recover the evolution of a software system using its *software trails*: information left behind by the contributors to the development process of the product, such as mailing lists, Web sites, version control logs, software releases, documentation, and the source code. This paper demonstrates the use of this method by recovering the evolution of Ximian Evolution, a mail client for Unix. By extracting useful facts stored in these software trails and correlating them, it was possible to provide a detailed view of the history of this project. This view provides interesting insight into how an open source software project evolves and some of the practices used by its software developers. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: large software systems; evolution; case study; open source software; historical data

1. INTRODUCTION

Investigating and recovering the evolution of a software project requires a combination of skills: it is necessary to understand the software product, its features, its components and how these have evolved; it is necessary to find, recover, and catalog valuable facts about the history of the project; it is required to look at the developing team, in order to better understand the software process they have used, their interrelations and communication, their decision taking and their skills; it also required to start piecing together all this information, proposing potential hypotheses that are then supported or not.

*Correspondence to: Daniel M. German, Department of Computer Science, University of Victoria, Victoria, BC, Canada V8W 3P6.
†E-mail: dmgerman@cs.uvic.ca

Contract/grant sponsor: National Science and Engineering Research Council/Advanced Systems Institute of British Columbia

Received 24 May 2003
Revised 10 January 2004
Accepted 21 April 2004

One can equate the work of a 'software evolutionist'[‡] to that of the combined efforts of a software architect, a historian, an ethnologist, an anthropologist, a paleontologist and a private investigator.

In rare cases, the evolution of a software product is recorded by an insider. This software evolutionist has access, presumably, to all the personnel and available information, and has the potential to accurately record its history as it unfolds. Unfortunately, few software projects have this type of resident historian and it is usually an outsider who has to do the work. This external software evolutionist could track the project for some time, looking from the outside at how the project continually evolves. Sometimes their work is done post-mortem, looking at the remnants of the project, like an anthropologist looking for clues of how an ancient civilization functioned. An outsider evolutionist depends on the available information of a project to tell its history. This paper defines *software trails* as pieces of information left behind by the contributors of a software project. Examples of software trails are configuration management systems logs, e-mail messages, documentation, recordings of conversations, product releases and, of course, the source code and other required files themselves. Software trails have the potential of keeping a 'community' memory of the software development. Linus Torvalds, for example, has repeatedly said that he would not have a telephone conversation to discuss the development of the Linux kernel, because he wants every decision to be recorded for posterity. The open source community recognizes that, given the volatility of core developers and an unforeseen future, keeping this information can provide important facts critical for the long-term survival of a project, taking it from one set of developers to the next and from one maintainer to another. Arguably, the best open source success stories tend to keep very detailed software trails. These trails can be used for two purposes: to educate future developers on the characteristics of the product, and to assist in recovering the history and evolution of the product. Closed source software projects are also interested in keeping this memory, as they know that their developing teams evolve with time and they cannot be dependent on one person maintaining this information in her head.

The rest of the paper is organized as follows. Section 2 describes a method to recover the evolution of a project using software trails and Section 3 details the methodology used to recover the evolution of the mail client Evolution using the described method. The paper continues with Section 4, which that describes the results in detail. Section 5 elaborates on some of the lessons learnt. The paper ends with conclusions and an outline of potential avenues for future work in Section 6.

2. RECOVERING THE EVOLUTION OF A PROJECT FROM ITS SOFTWARE TRAILS

This paper proposes a method of recovering the evolution of a software system by analyzing the software trails left behind during its development. This method is composed of four steps.

1. *Define schema*. Create a schema that represents the information available in the software trails, including any relationships between them. For example, that a developer has a list of different e-mail addresses used to post to the mailing lists; that a particular defect was fixed by a given

[‡]A software evolutionist is a person who studies how software evolves. The author believes that there is no term in the literature that denotes this role.

developer with a given set of software changes; that a software change includes a delta of the change and a version number; etc.
2. *Gather software trails*. Retrieve the available software trails and map them into this schema. Often the logs of these trails are not easy to parse nor translate. In some cases heuristics need to be developed and applied.
3. *Extend information*. The available software trails can be extended by further analysis, enhancing them by extracting new facts or creating new relations as is appropriate. For example, many open source developers do not use configuration management software and the developer usually informally states (in the version control log) that a given set of changes corresponds to a defect fix; the version control log has to be parsed in order to find something that 'might look' like, its corresponding defect number.
4. *Analyze*. The final step is to look through this data and try to find interesting events in its development that can tell the history of the project. This is a difficult problem. As the software evolves and grows, its available information grows at the same time, making it difficult for the evolutionist to find the 'more' relevant information that tells them an interesting fact about the history of the project.

Given the informal nature of some of these trails (and the fact that it is a reverse engineering process, where the evolutionist has no certainty on what were the actual events and they are merely trying to reconstruct them from the trails available), the experience and insight of the evolutionist and amount of time that they invest in the analysis of the information available will have an important impact on the quality of the results. In order to demonstrate the above methodology, this paper uses the software trails of Ximian Evolution to recover its evolution. Evolution is a mail client (similar in scope to Microsoft Outlook) that is starting to gain popularity in the Unix world. Evolution developers have left software trails in mailing lists, Web sites, its CVS repository logs (CVS is one the most widely used version control systems), documentation, inside and outside the code, and Bugzilla, its bug-tracking system.

Different software trails have been used extensively in the literature to help in the understanding and modeling of software. For example, Godfrey and Tu used primarily releases to recover the evolution of the Linux kernel [1], while configuration management data were used in [2–6].

3. METHODOLOGY

The following software trails were used in the recovery of the evolution of Evolution.

- *Version source code releases*. As of May 2003, there have been 37 different releases. These come in the form of tar files that contain all the necessary files to build and run the product. They are made available for the people who are interested in recompiling the product to suit their particular installation. Five of these releases are considered major (0.0, 10 May 2000; 1.0, 21 November 2001; 1.1.1, 9 September 2002; 1.2.0, 7 November 2002; and 1.3.1, 28 February 2003). Evolution has adopted a numbering scheme similar to the Linux kernel, using odd numbers in the second component of a release label (such as 1 or 3 in 1.1.1 and 1.3.1, respectively) to denote 'unstable' releases that are considered to be riskier (buggier) than the stable ones (such as 1.0 and 1.2.0). Source code releases can be seen as a coarse-grained view of the evolution of a project.

A collection of scripts and tools such as 'exuberant ctags' and 'stripcmt' were used for fact extraction (similar to the fact extraction in [1]).

- *CVS logs*. CVS keeps track of who modifies which file and the corresponding delta associated with the modification. This change is known as a 'file revision'. CVS keeps information such as who made the revision, when, the diff of the revision, the number of lines added and the number of lines removed (Fisher *et al*. described some of the challenges of extracting information from a CVS repository in [7]). `softChange` [8] was used to recover the information from these logs and to enhance it. For instance, CVS does not keep track of which files are modified at the same time. `softChange` analyses the logs and rebuilds these groups of files, which are then called modification requests (MRs). A MR is a request by a contributor to commit a group of files at the same time. The belief is that if two files are part of the same MR, it is because they are somehow interrelated. In contrast to source code releases, CVS logs provide a very fine-grained view of the evolution of the project. A snapshot of the CVS log was taken on 21 May, 2003.
- *Mailing lists*. Evolution maintains at least two mailing lists, but some of the information related to can also be found in the GNOME mailing lists. GNOME (GNU Network Object Model Environment) is a free software collection of libraries and end-user applications that provide a graphical 'desktop' for Unix systems. GNOME is the parent project of Evolution. Mailing lists tend to serve as a record of important decisions related to a project. Another use of mailing lists is to announce the availability of new releases (including a summary of its new features).
- *ChangeLogs*. As the GNU ChangeLog standards indicate, the ChangeLog explains how earlier versions of software were different to the current version.

For the purpose of this paper, `softChange` was extended to generate relational data, which was then imported into a `postgresql` database (the dump of the database measures 0.5 Gbytes, although some tables contain redundant information to help speed-up queries—a copy of the database is available on request and from turingmachine.org/evolution). The analysis of the data was done *ad hoc*, writing SQL queries. SQL was very helpful in filtering and tabulating information, that could then be plotted (our research team has since developed a tool to automatically create many of the plots displayed herein using SVG using the Web as its interface).

4. EVOLUTION

This section presents a detailed view of the evolution of Evolution. It starts by giving a brief history of the project. It then provides an analysis of its releases, giving a coarse-grained view of how the product has evolved. It then proceeds to get a more detailed view of the development process by looking at the developers activity and the typical contributions they make to the product. If finalizes by looking in detail at the way the product has been divided into modules, and the implications that such modularization has had in the software development process.

4.1. History

At the beginning of 1999, Bertrand Guiheneuf started working on a new mail client for the GNOME project [9]. One of his goals was to create a better mail client than Balsa (the GNOME mail client

at the time) and to use Bonobo (GNOME CORBA implementation) to display the different content types in e-mail messages. He decided to start the project by implementing a mail storage library, which he called *camel*. In Guiheneuf's view, Balsa was not good enough. He planned, however, to phase in the development of camel by incorporating its storage library into Balsa (and other potential mail clients) using CORBA [10]. The *GNOME Mailer* project was formally started in 16 April, 1999 with a mail message from the GNOME project leader Miguel de Icaza that discussed the need for a more powerful mail client [11]. One important issue that de Icaza addressed in this message was why not further develop an already started project (such as Balsa)? His answer was 'there is too much baggage in existing mail applications that we do not want to carry into the future'. This message was probably triggered by Guiheneuf's posting (two weeks before).

de Icaza, following Guiheneuf's ideas (and the trend of GNOME in general), proposed the use of CORBA for communication between these modules and other applications that would help display different content types in the message display (at the time, there was a move towards making most GNOME applications CORBA aware). This module list would also serve as a way to divide the work into pieces in which different developers could concentrate and work as independently as possible. A mailing list was created for the project, and during April 1999 more than 500 messages were exchanged, most of them related to requirements analysis for the new project. Guiheneuf would become the first maintainer of the new GNOME Mailer, continuing the development of camel as its storage module. In August 1999, the name Evolution was proposed by him, and it was quickly accepted by the GNOME community[§]. In October 1999, de Icaza created Helix Code (now Ximian), a commercial venture aimed at continuing the development of GNOME, planning to generate income by selling services around it. Ximian proceeded to take under its wing the development of Evolution and has committed several employees to work on it. In four years Evolution has grown into a powerful product that is starting to be widely used in the open source community. Evolution recently received the '2003 LinuxWorld Open Source Product Excellence Award' in the category of 'Best Front Office Solution'. Ximian was bought by Novell in August 2003.

4.2. Releases

Figure 1 shows the growth in the size of the source code releases of Evolution. It was discovered that the total size of the release (sum of the size of all files) and the total size of the source code (sum of the size of all source code files) did not show a clear correlation. Further investigation demonstrated that the main culprit for the increase of the size of the release is its internationalization (translation files with extensions .po and .gmo). The latest version, for example, totals 64 Mbytes of which 37 Mbytes (57%) are internationalization files, compared with only 11 Mbytes of source code (17%). Evolution is currently translated into 34 different languages (this does not include regional variants; for example, Evolution includes internationalization files for Portuguese and its Brazilian variant). Another surprise is to discover that the next largest contributor to the size of a release is ChangeLogs: 4.6 Mbytes (7%). ChangeLogs will be discussed further below.

[§]Guiheneuf proposed e-volution, which was quickly altered to evolution. The name was later changed to Evolution and finally to its current official name Ximian Evolution.

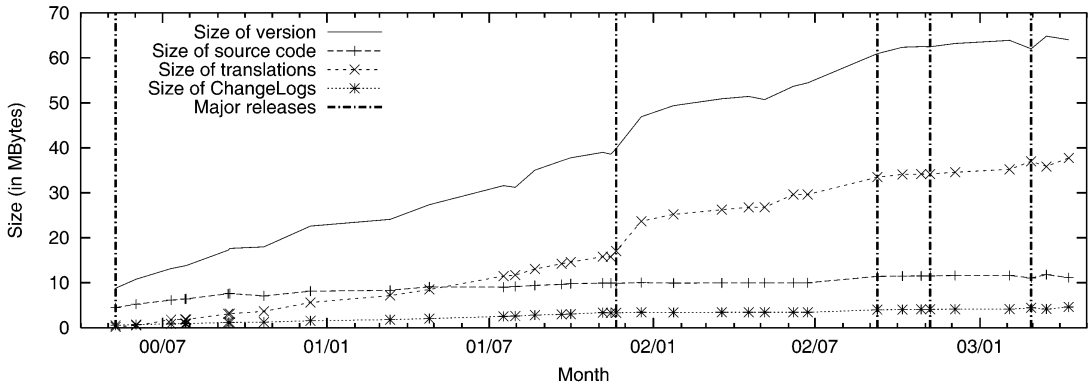

Figure 1. Size of releases over time.

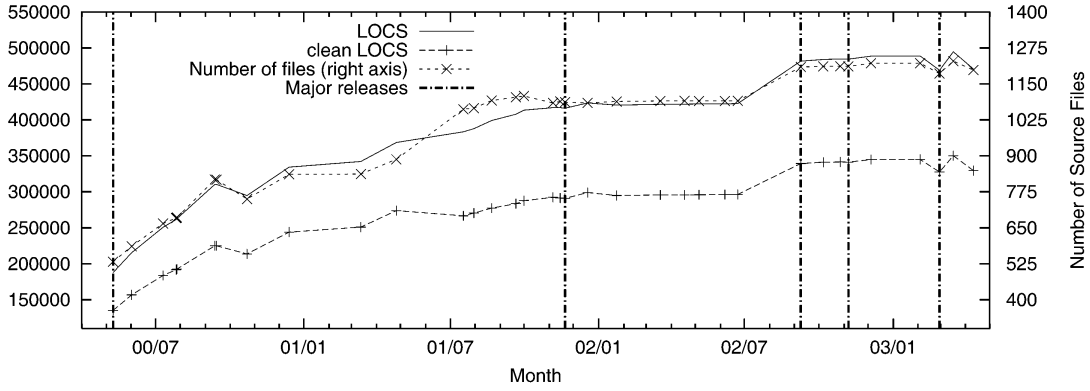

Figure 2. Evolution of source code in releases.

The number of files shows a different picture. The average proportion of source files in the releases is 46% (6.16 σ (standard deviation)). In contrast, the proportion of translation files is 2.7% (0.29 σ) and 1.1% (0.03 σ) for ChangeLogs. Translation files and ChangeLogs are therefore few, but very large files when compared to source code files.

Figure 2 shows the number of source code files, total source lines of code (LOCS) and total cleanLOCS (number of LOCS when comments and empty lines have been removed) for a given release. The average size of a source file has been stable across versions, at 639 (25 σ) LOCS per .c and 101 (7.6 σ) LOCS per .h file. The proportion of cleanLOCS to LOCS has also remained stable across versions, at 72.5% (1.4 σ) for .c files and 60% (2.6 σ) for .h files.

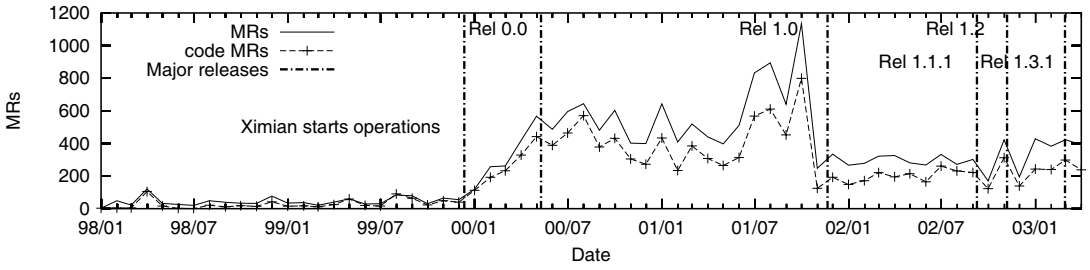

Figure 3. Evolution of the project in number of MRs.

The actual change in LOCS from one version to another shows an interesting story. Of special interest are the negative increments in either LOCS or source files, suggesting removal of source code. For example, in version 0.6 (released 23 October 2000) 15 500 LOCS and 67 source code files were removed with respect to the previous version (0.5.1). Between these two releases, 157 source code files were deleted and 90 created (45 000 LOCS were deleted and 24 000 LOCS added). Further analysis of the available software trails showed that for this release it was decided that several widgets should be moved (from Evolution's graphical user interface) to the Gal project. Gal, according to its official description is 'the GNOME Application Library, a collection of widgets and other helper functions originally extracted from Evolution and gnumeric (GNOME spreadsheet)'. In fact, the first version of Gal (0.1) was released on 5 October 2000 [12], five days before Evolution 0.6 (and the sudden drop in LOCS).

4.3. Development activity

One important question that arises when looking at the increment in the size of Evolution is how does it correlate to the actual activity of the developers? The CVS logs provide some useful information that can be used in an attempt to answer this question.

Figure 3 shows the number of MRs per month for Evolution. The plot also shows the major releases of the project. There are several interesting observations from this graph. First, the development activity was relatively flat during the first year of the development and it is not until Ximian is born that there is a surge in the number of MRs. The number of MRs surges just before release 1.0. After that, the number of MRs remains more stable, but still shows peaks that correspond to releases. Because it is not possible to have access to the actual number of hours spent per developer in the project, it is not possible to determine the development effort spent per MR and, therefore, whether fewer MRs mean less developer time or if some MRs required more time. In the same figure, the number of MRs that involve source code (codeMRs) is also shown. The proportion of codeMRs to MRs has decreased during 2003 (approximately 38% of the MRs do not involve source code).

Why has the proportion of codeMRs dropped? The exploration of the logs drew the following conclusions. From all of the MRs in 2003, 18% were changes to metafiles, 13% corresponded to internationalization, while only 61% corresponded to changes in source code (some MRs included

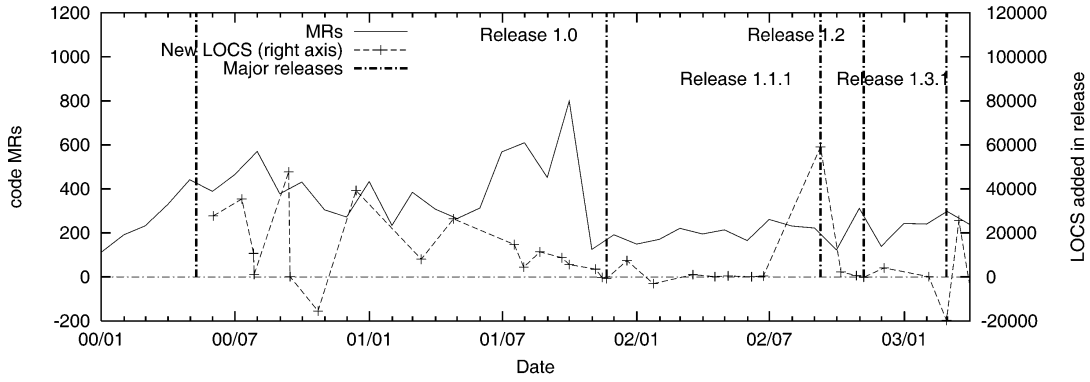

Figure 4. Changes in LOCS and number of files per version.

changes to metafiles and source code, and some MRs included changes to metafiles and translations). Metafiles are used by the `automake` and `autoconf` tools to create other files. The most common use of these metafiles is the creation of Makefiles (the developer creates an .am or .in file, and `autoconf` and `automake` create the corresponding Makefile). Metafiles rely heavily on macros (GNOME provides a module called macros with the majority of these definitions). A surge in the activity related to metafiles and translations was to blame for the drop in the proportion of codeMRs. The question that followed was, what prompted the surge in metafile activity? In those MRs 70% of the revisions corresponded to Makefile.am files and 12% of the revisions corresponded to changes to `configure.in`, the main `autoconf` file that drives the configuration of Evolution when a user wants to compile it. Inspection of the ChangeLogs seems to suggest a conscious effort to cleanup the metafiles. The surge in changes to the translations is attributed to a previous significant change in the user interface. Once the development team decides to make a 'freeze' in the features of a release, translators start making changes to the corresponding translation files. Another question prompted by Figure 3 is why does it show activity before January 1999? It appears that some code that was in development previous to Evolution was later incorporated into it (one widget and some calendar related code). It was also noted that some revisions contain invalid dates, suggesting that during a period of time the clock of the CVS server was set to an incorrect time.

Figure 4 shows codeMRs and how they relate to the actual growth in the size of the source code in the releases. Even in periods where the code base does not increase (like the first half of 2002) the number of MRs is still significant. This suggests a period in which debugging took precedence over the development of new features. It is also interesting to see the typical characteristics of an MR. Figure 5 shows the number of files per MR (to clarify the distribution of data in Figures 5–8 and 10, in which significant portions of points are concentrated at either end of the scale, we decided to use logarithmic scales where appropriate). 75% of them contain three or less files, which is a healthy sign. The log for the largest MR (which contains 650 files from 23 June 2001) reads 'Update the copyrights, replacing Helix Code with Ximian and helixcode.com with ximian.com all over the place'. That day a total

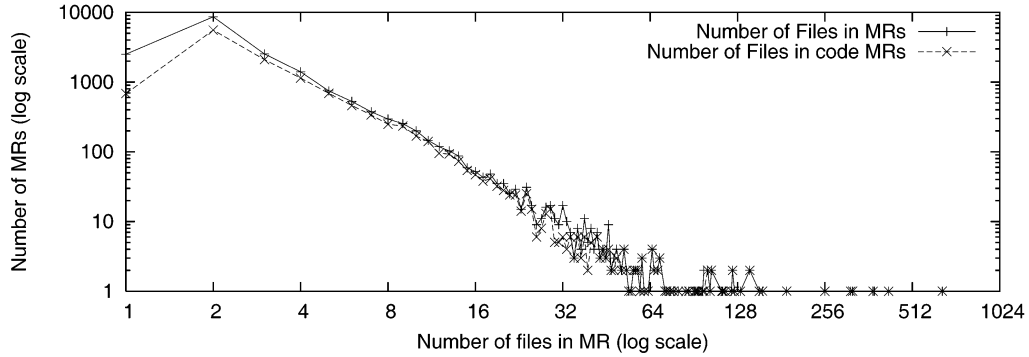

Figure 5. Number of MRs that modify a given number of files.

of 709 files were modified. Similarly, the largest number of files modified in a single day was 1417 (27 October 2001) and the reason was 'update the licensing information to require version 2 of the GPL (instead of version 2 or any later version)'. These two explanations highlight a particular feature of MRs in Evolution: developers take care to explain, in each MR, the reason for the change (CVS allows developers to add a log message to every file revision during a CVS commit). The average log for an MR is 300 characters (561 σ, 170 median), with a minimum length of 1 (only 8 MRs) and 18 000 for the longest log (which involved the merging of a branch to the main CVS tree).

There is a common belief that large open source projects are developed by teams composed of several hundred individuals. It is, however, important to recognize that the contribution of the majority of these individuals is very small. In open source projects, contributors can be divided into two main groups: those with write access to the CVS repository (and who can make their contributions to the CVS repository themselves) and those who do not have write access to the repository. In GNOME it is not difficult to get write access to the repository. Once somebody has submitted several contributions, this person can apply for CVS write access. In GNOME, more than 500 people have CVS write access.

By looking at the changes committed by contributors with CVS write access, we can see that, like many other open source projects, the majority of the coding is done by just a few individuals. Zawinsky, once one of the core Mozilla contributors, commented on this phenomenon: 'If you have a project that has five people who write 80% of the code, and a hundred people who have contributed bug fixes or a few hundred lines of code here and there, is that a 105-programmer project?' (as cited in [13]).

Evolution contains contributions by 201 different userids (contributors). Relatively few of these, however, contributed a significant portion of the MRs. For example, 48% of all the MRs were contributed by only five contributors, while 142 contributors contributed just 5% of the MRs. Figure 6 shows the proportion of MRs per contributor (each contributor was assigned a number from 1 to 201, which corresponds to the x-axis). Only 18 contributors accounted for more than 1% of the total MRs. The most active contributor is responsible for 16% of the MRs.

Table I shows the 10 most active contributors, as a proportion of all MRs. The top 9 appear to be Ximian employees or consultants (see http://primates.ximian.com/). These 10 individuals were

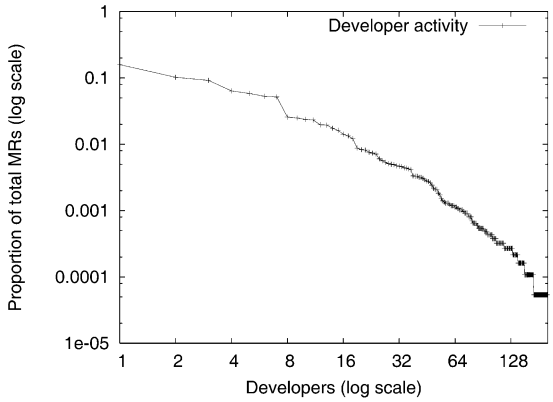

Figure 6. Proportion of MRs per contributor.

Table I. Most active contributors, as a proportion of total MRs.

Userid	Proportion of total	Accumulation
fejj	0.16	0.16
ettore	0.10	0.26
danw	0.09	0.35
zucchi	0.06	0.42
clahey	0.06	0.48
jpr	0.05	0.53
toshok	0.05	0.58
federico	0.03	0.61
peterw	0.02	0.63
iain	0.02	0.65
other	0.35	1.00

responsible for 65% of the total MRs. This fact corroborates the hypothesis that private companies (such as RedHat, Ximian and Eazel) have had a very important effect on the development of the GNOME project [14]. In that respect it is similar to the Mozilla project, where core contributors were employees of Netscape (see [5]).

How regularly were contributors participating in the project? After January 2000 there was an average of 32 contributors (8.3 σ, minimum 15, maximum 47) per month to the project.

4.4. Revisions

Every time a file is modified, CVS creates a record of who modifies it, when, and the 'delta' of the modification. This modification is known in CVS lingo as a 'revision'. Table II shows the proportion

Table II. Revisions and number of files in CVS per file extension.

Extension	Proportion of total	Accumulation	Files
.c	0.41	0.41	1195
ChangeLog	0.22	0.62	43
.h	0.13	0.75	1063
.am	0.05	0.81	174
.po	0.04	0.85	71

Table III. Top 10 most modified files.

File	Proportion of total	Accumulation
mail/ChangeLog	0.04	0.04
calendar/ChangeLog	0.03	0.06
camel/ChangeLog	0.03	0.09
addressbook/ChangeLog	0.02	0.11
shell/ChangeLog	0.02	0.13
ChangeLog	0.02	0.14
po/ChangeLog	0.02	0.16
configure.in	0.01	0.17
composer/ChangeLog	0.01	0.18
mail/mail/callbacks.c	0.01	0.18

of revisions per extension (i.e., type of file). Given that C is the language of choice for Evolution, it is not surprising to see .c and .h files at the top, along with ChangeLogs; `make` metafiles (.am) and internationalization files (.po) follow.

ChangeLog files are an important source of information about the development and evolution of a project. The Evolution developers are fairly consistent in their modifications to the ChangeLog files. From all MRs involving two or more files, 93% include a modification to a ChangeLog. Evolution developers seem to make sure that they document their changes in the corresponding ChangeLogs. Table III shows the 10 most modified files, eight of which are ChangeLogs. ChangeLogs (and CVS logs) can provide insight on patches submitted by developers without a CVS account, as developers are expected to be careful to give credit to the patch submitter in the corresponding ChangeLog entry (which are not taken into account for this paper).

In terms of source code hot spots, there have been a total of 41 120 revisions to 2258 source code files (many of these files are no longer in the latest release, as they have been removed during the development process, CVS keeps information about their modification). Figure 7 shows the proportion of revisions per source code file. 51 files account for 25% of the total number of revisions, while 764 account for only 5% of them.

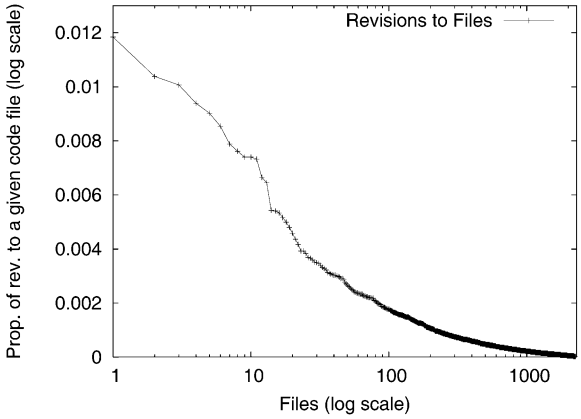

Figure 7. Proportion of revisions per source code file.

4.5. Modularization

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 [15]. From the beginning of the project, there has been a conscious attempt to divide Evolution into modules that fulfill the previous characteristics. Modules are represented in the code base as subdirectories. Figure 8 shows the different modules and the number of MRs for each of them, representing the level of activity in each module. Figure 9 shows the size of the seven largest modules in Evolution in terms of LOCS. With the exception of libical and widgets, modules tend to grow in size over time. Before 2002, both libical and widgets saw an increase and decrease in their LOCS. Since Version 1.0, the size of Evolution has been growing at a much smaller rate than previously.

Other interesting questions remain. Do contributors tend to concentrate in one module? How many active core contributors does a given module have? Some Evolution core developers have left the project and new ones have taken their place. In an attempt to isolate the developers who are still active in the project, we decided to only look at the data from 2002 (the last complete year available when this research took place). Table IV shows the most active developers for the four most modified modules of Evolution during this year. It is not surprising to see that one or two contributors are responsible for at least two thirds of the MRs in each module.

Finally, how well do modules isolate developers from the complexity of other modules? One potential way to measure this dependency is to analyze the number of codeMRs that require changes in more than one module. Figure 10 shows a compelling story: only 3% of the codeMRs include files from more than one module. Further analysis of the changes is required to determine the proportion of changes that were actual code changes rather than simple changes to comments.

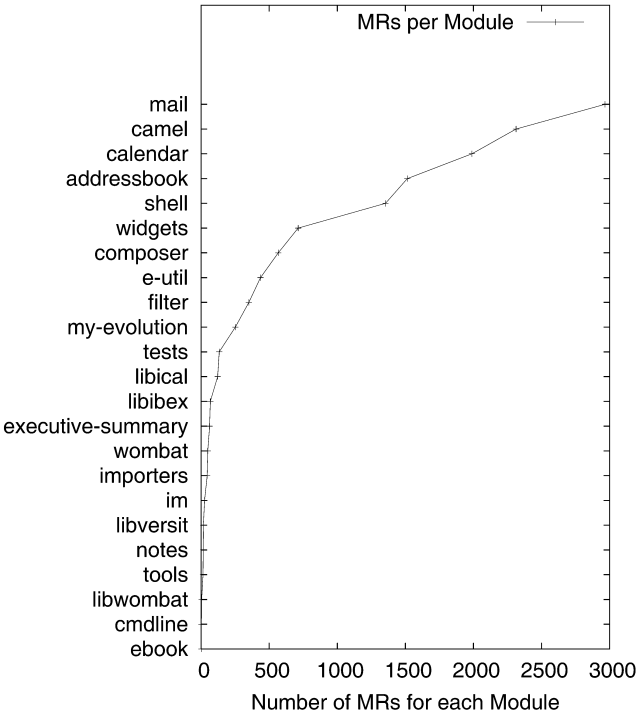

Figure 8. MRs per module: most of the activity is concentrated on few modules.

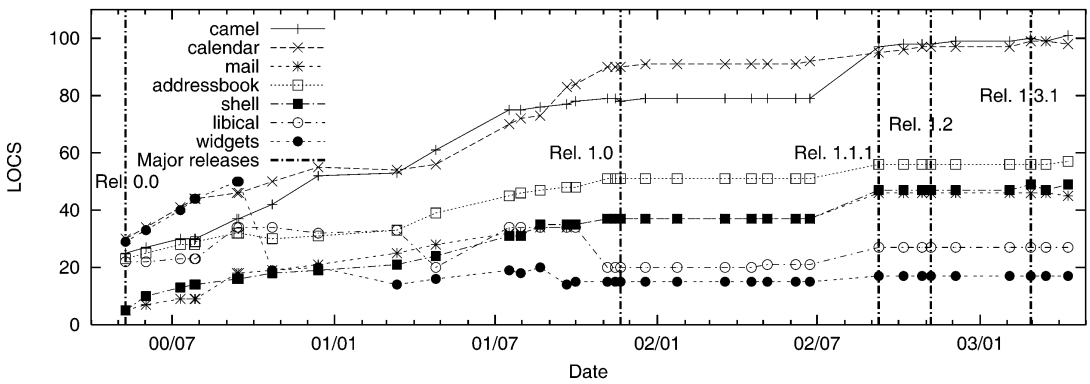

Figure 9. LOCS in selected modules per version.

Table IV. Top 5 programmers of some the most active modules during 2002.

Module	Number of programmers	Id	Proportion of total	Accumulation	Module	Number of programmers	Id	Proportion of total	Accumulation
calendar	17	jpr	0.40	0.40	addressbook	19	toshok	0.57	0.57
		rodrigo	0.32	0.72			clahey	0.13	0.70
		ettore	0.07	0.79			ettore	0.09	0.79
		danw	0.06	0.85			danw	0.07	0.87
		damon	0.03	0.88			fejj	0.03	0.90
mail	19	fejj	0.52	0.52	camel	9	fejj	0.66	0.66
		rodo	0.13	0.65			zucchi	0.25	0.91
		zucchi	0.12	0.77			danw	0.03	0.94
		ettore	0.07	0.83			peterw	0.03	0.97
		danw	0.06	0.89			ettore	0.01	0.99

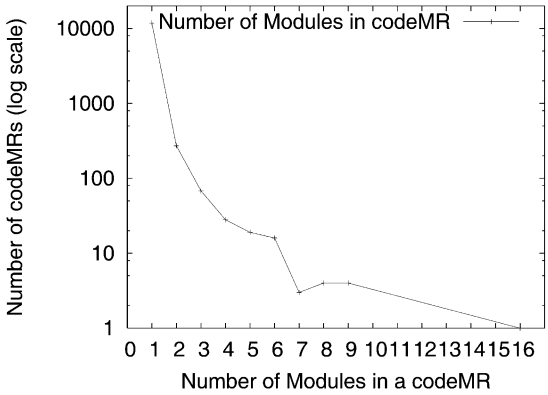

Figure 10. Number of different modules that appear in a codeMRs.

5. FURTHER OBSERVATIONS

The above results show that the method described in this paper can be applied to recover the evolution of a software project as long as the number of existing software trails is sufficient for the analysis. Several observations can be made about this experience.

The analysis of software trails can be done with at least two different purposes in mind. First, in a focused approach, the analysis can be used to answer a specific set of questions about the project. Second, with an open-ended approach, the trails can be analyzed with a view toward uncovering interesting facts about the project. This research started with some specific questions in mind [14], for example, how important were the contributions made by Ximian employees to the project, how the activity of the developers correlated to the release of new versions and how good the ChangeLogs were. With time, the focus of the research changed to an open-ended approach. There were many interesting facts worth pursuing and reporting on, and many questions arose as more facts were uncovered. At the end, both approaches proved to be useful towards the recovery of the evolution of the project.

One software trail does not tell the whole story. It is paramount to cross-reference software trails to really understand what they mean in the evolution of the project. For example, the growth in the size of the released software for Evolution—which, as explain before, includes more than just source code—is not proportional to the growth in the size of its source code; also, many developers have been participating in the project, but most of them with very few contributions.

The schema used in this study kept changing, in part because of the incorporation of new trails, and also because new information and relations were discovered. It is expected that, as this type of analysis becomes more pervasive, standard schemas can be developed. This will have two advantages: it will promote the creation of tools that gather software trails, extend them and analyze them; and the evolutionist will better understand the nature of the available trails before starting their work.

One of the main challenges of analyzing software trails is that many of them are informal in nature. For example, e-mail messages contain a significant amount of information pertaining to the way the

project has evolved, but they are difficult to analyze in an automatic fashion. Correlating different trails is also an error prone task, in which heuristics have to be developed and tested. It might be the case that a heuristic performs differently in different projects.

The amount of information available from the software trails makes use of the visualization and filtering tools indispensable in their analysis. Again, as schemas are standardized, different research teams could provide different tools that specialize in mining and visualizing certain types of trails. In this paper, SQL was chosen because it provides a sophisticated query language (further extended in `postgresql` with its support for regular expressions in the `where` clause). It is also interesting that Evolution itself proved very useful in analyzing the Evolution mailing lists, given that it provides a powerful query language for e-mail messages.

It is important to state that not all development teams generate 'good' software trails. In the experience of the author, there is a point in a software project in which software trails start to 'mature' and this point is likely a correlation of the success of the project, the level of interaction that developers have to have and their maturity, and in the case of commercial projects, the influence of management. For instance, there is very little information about Evolution when only one developer was contributing to it, but as the developers grew in number (and became more experienced) their trails improved in quality. The Free Software Foundation has an important effect in the quality of trails, as it publishes a collection of guidelines that free software developers should follow.

This paper hopes to provide, on one hand, a view on how an open source project evolves and its developers interact. This information could be useful when researchers try to understand how open source software development takes place. On the other hand, it is also hoped that with the insight gained by looking at the details of several software projects it will be possible to come out with more automatic and formal processes that can be easily applied to any project, with little manual intervention. By themselves the results presented in this paper might be of little relevance, but they might provide valuable information towards the creation of a formal model for the automatic analysis of software trails.

6. CONCLUSIONS AND FUTURE WORK

This paper demonstrated a methodology to recover the evolution of a software project using its software trails. Software trails, such as version releases, version control logs and mailing lists were used to recover the evolution of Ximian Evolution, a free mail client for Unix. The analysis of these software trails allowed the discovery of interesting facts about the history of the project: its growth, the interaction between its contributors, the frequency and size of the contributions, and important milestones in its development. The analysis of this open source project permits us to make some observations with respect to Evolution.

- The size of a distribution grows faster than the size of its source code, as internationalization and documentation files tend to grow faster than the code base.
- Although the number of people who have contributed to the project is large (more than 200), most of the work is done by a small number of contributors (10 contributors account for 65% of the total MRs).
- Most of the programmers are people employed to contribute to the project.

- Internationalization is one area in which sporadic contributors (usually volunteers) tend to concentrate.
- Developers tend to concentrate on one module, therefore reducing the amount of communication and interaction with other developers.
- Most MRs contain a very small number of files (75% of the MRs contain three or less files), and they usually include a detailed description of the modification in the ChangeLog files.

These observations need to be compared against other open source projects, in order to draw some general conclusions with respect to how these projects are developed. There are several other potential avenues for future research. One of them is to create tools that analyze and enhance the facts that were extracted. For example, CVS's MRs can be analyzed in an attempt to guess the type of modification that the developer intended: a comment, a bug fix, a new feature or refactoring. This will allow the evolutionist to quickly categorize changes and concentrate on those of interest. Another area of research is the visualization of this information. As the project grows older, its trails grow in number. It is necessary to create tools that analyze and display the gathered facts to the user and allow its visualization in a highly dynamic manner. Some work has been started in this direction, for example, Cubranic and Murphy have proposed a tool to query software trails [16], while Fisher and Gall have proposed an environment to visualize the interrelations between files in a project using their CVS software trails [17].

Metrics are also an important area of research. The information extracted from software trails must be quantified, so it can compared with other software projects. For example, how can the MR activity of one project be characterized and compared against the activity of another project? Finally, studies on other software projects (similar to that in this paper) are needed. These studies will provide information necessary to better understand the characterization of software trails. Furthermore, these studies will allow researchers to compare the evolution of different software projects and, to a certain extent, some of the practices used by their respective development teams.

ACKNOWLEDGEMENTS

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 Evolution development team, Audris Mockus (co-author of `softChange`) for his invaluable help in a preliminary analysis of Evolution and especially the anonymous reviewers whose helpful and detailed comments made this a better paper.

REFERENCES

1. Godfrey MW, Tu Q. Evolution in open source software: A case study. *Proceedings International Conference on Software Maintenance (ICSM 2000)*. IEEE Computer Society Press: Los Alamitos CA, 2000; 131–142.
2. Atkins D, Ball T, Graves T, Mockus A. Using version control data to evaluate the impact of software tools: A case study of the version editor. *IEEE Transactions on Software Engineering* 2002; **28**(7):625–637.
3. Eick S, Graves T, Karr A, Mockus A, Shuster P. Visualizing software changes. *IEEE Transactions on Software Engineering* 2002; **28**(4):396–412.
4. Gall H, Hajek K, Jazayeri M. Detection of logical coupling based on product release history. *Proceedings International Conference on Software Maintenance (ICSM 1998)*. IEEE Computer Society Press: Los Alamitos CA, 1998; 190–198.
5. Mockus A, Fielding RT, Herbsleb J. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology* 2002; **11**(3):1–38.

6. Mockus A, Votta LG. Identifying reasons for software changes using historic databases. *Proceedings International Conference on Software Maintenance (ICSM 2000)*. IEEE Computer Society Press: Los Alamitos CA, 2000; 120–130.
7. Fischer M, Pinzger M, Gall H. Populating a release history database from version control and bug tracking systems. *Proceedings International Conference on Software Maintenance (ICSM 2003)*. IEEE Computer Society Press: Los Alamitos CA, 2003; 23–23.
8. German DM, Mockus A. Automating the measurement of open source projects. *Proceedings of the 3rd Workshop on Open Source Software Engineering*. University College Cork: Cork Ireland, 2003; 63–68.
   Available at: http://opensource.ucc.ie/icse2003 [April 2004].
9. Guiheneuf B. *Candidate Bertrand Guiheneuf (GNOME Foundation Announce Mailing List)*. The GNOME Project: Boston MA, 2000. Available at: http://mail.gnome.org/archives/foundation-announce/2000-October/msg00009.html [April 2004].
10. Guiheneuf B. *GNOME Mail Clients Re: Is Balsa alive? (GNOME Developers Mailing List)*. The GNOME Project: Boston MA, 1999. Available at: http://mail.gnome.org/archives/gnome-devel-list/1999-April/msg00042.html [April 2004].
11. de Icaza M. *Writing a GNOME Mail Client (GNOME Announce Mailing List)*. The GNOME Project: Boston MA, 1999. Available at: http://mail.gnome.org/archives/gnome-announce-list/1999-April/msg00029.html [April 2004].
12. de Icaza M. *G Apps Lib 0.1 is Out (GNOME Announce Mailing List)*. The GNOME Project: Boston MA, 2000.
    Available at: http://mail.gnome.org/archives/gnome-announce-list/2000-October/msg00005.html [April 2004].
13. Jones P. *Brooks' Law and Open Source: The More the Merrier? Does the Open Source Development Method Defy the Adage About Cooks in the Kitchen? (IBM developerWorks)*. IBM Corporation: White Plains NY, 2000.
    Available at: ftp://www6.software.ibm.com/software/developer/library/merrier.pdf [April 2004].
14. German DM. The evolution of the GNOME Project. *Proceedings of the 2nd Workshop on Open Source Software Engineering*. University College Cork: Cork Ireland, 2002; 20–24.
    Available at: http://opensource.ucc.ie/icse2002 [April 2004].
15. Lerner L, Triole J. The simple economics of open source. *Working Paper 7600*, National Bureau of Economic Research, Cambridge MA, 2000. Available at: http://www.nber.org/papers/w7600.
16. Cubranic D, Murphy GC. Hipikat: Recommending pertinent software development artifacts. *Proceedings of the 2003 International Conference on Software Engineering (ICSE 2003)*. ACM Press: New York NY, 2003; 408–418.
17. Fisher M, Gall H. Visualizing feature evolution of large-scale software based on problem and modification report data. *Journal of Software Maintenance and Evolution: Research and Practice* 2004; **16**(6):385–403.

AUTHOR'S BIOGRAPHY

Daniel M. German is assistant professor at the University of Victoria, in Canada. His areas of research are the design of hypermedia applications, the formal specification of hypermedia and software systems, and the evolution of open source software. He obtained his PhD from the University of Waterloo in 2000.

