Studying the evolution of libre software projects using publicly available data

Gregorio Robles-Martínez, Jesús M. González-Barahona,
José Centeno González, Vicente Matellán Olivera, and Luis Rodero Merino
GSyC, Universidad Rey Juan Carlos
{grex,jgb}@gsyc.escet.urjc.es

Abstract

Libre software projects offer abundant information about themselves in publicly available storages (source code snapshots, CVS repositories, etc), which are a good source of quantitative data about the project itself, and the software it produces. The retrieval (and partially the analysis) of all those data can be automated, following a simple methodology aimed at characterizing the evolution of the project. Since the base information is public, and the tools used are libre and readily available, other groups can easily reproduce and review the results. Since the characterization offers some insight on the details of the project, it can be used as the basis for qualitative analysis (including correlations and comparative studies). In some cases, this methodology could also be used for proprietary software (although usually losing the benefits of peer review). This approach is shown, as an example, applied to Mono, a libre software project implementing parts of the .NET framework.

1 Introduction

Since its birth, software engineering has been trying to gain knowledge on the software development process in order to quantify the timing, human costs and technical resources that lead to a successful software development. Even so, in most cases experience has been acquired by studying in detail a handful of software projects that were accessible only to the researcher doing the study (due to intellectual propriety constraints). Important facts, like the detailed evolution of the source code or the developers working in the project at any given time were not publicly available for peer review. Quantitative characterization of those projects were, usually, rather incomplete, making it difficult to compare and correlate data.

However, since several years ago we have at our disposal a good quantity of data about thousands of libre (free, open source) software projects. Most of them provide the researcher with very rich and complete information about their state at any given time[1]. Being the number of projects and data about them so huge, it seems important to use a consistent and as much automated as possible methodology to go from the data available to the characterization. This should make possible the analysis of a large fraction of the available information, and simplify comparative analysis.

On the other hand, focusing on libre software engineering, there is little work on the quantitative characterization of libre software projects[2]. To be able of getting some conclusions about libre software development, a lot of work on merely retrieving data about the projects has to be done. Fortunately, it can be performed on an uniform and semi-automated way.

From the different data available for libre software projects, we propose a methodology based on the analysis of the source code in CVS repositories, from three different points of view: source code size, interaction with the versioning system and authorship information. The combination of these approaches provide a detailed and complete picture of the project and its historical evolution.

2 Tools and methodology

The proposed methodology is based on the use of CVS[3]. Fortunately, there are a lot of projects in this situation: for

[1] In addition to the source code itself, several systems are used which store historical and well structured information. The most relevant among them is CVS (and its derivatives), but there are others: bug tracking systems, mailing lists, release snapshots, etc.

[2] There are several well known papers trying to cast libre software projects from a more qualitative point of view (being [12] probably the most popular), but they have to face a lot of criticism due to the lack of quantitative data and methodology on which to base the analysis [1]

[3] Concurrent Version System, the versioning system most popular in the libre software community. Many libre software projects use a repository where all the source code and documentation are stored. Developers interact with the central repository by checking out the latest version, modifying it and committing back the changes. With every commit some information is stored in the CVS log files: the committer, the date, the lines that have

instance, more than 10,000 projects hosted at SourceForge use it [7], and the 11 largest projects in Debian 2.2 [5] use versioning systems (all of them CVS except for Linux that uses Bitkeeper, a proprietary solution). The core tools used to study the data in the CVS repository are the following ones[4]:

- **CODD** [2] searches for authorship information in source code, by tracking copyright notices and other information in the headings of files. It then assigns the length (in bytes) of each file to the corresponding authors. CODD has been used in several studies since its first release in 1999 (Orbiten Survey [3], source code survey done for the FLOSS study [4], etc.)

- **SLOCCount** [13] counts the number of physical source lines of code (from now on, SLOC[5]) for a given directory. It can recognize dozens of different languages, and tell apart files with source code from those with documentation or other data. SLOC counts are the input needed by the Basic COCOMO model (also known as COCOMO I) to estimate minimum schedules, efforts and costs. SLOCCount has been used to study large collections of software, like GNU/Linux distributions such as Debian [5] and RedHat [14].

- **CVSanalY** allows the statistical analysis of CVS. It gets the history of every file in the repository, and determines who did what for each commit.

- **GlueTheos** has been developed to coordinate the other tools to implement the methodology. It extracts snapshots from the CVS repository at several points in the past, and use the other tools to analyze them. It also normalizes information in an XML format, suitable for use as a detailed description of a project. From that format, it can get statistical data, graphical information and other formats (for instance, SQL tables) for further analysis.

When studying a project, GlueTheos drives the other tools to analyze it. It runs CVSanalY to get the historical data directly from the CVS repository, looking for the interactions of the developers with the repository. It also checks out periodic snapshots of the source code, and runs on them SLOCCount and CODD to get information on the authors and sizes of the code. It later uses the combination of all this information to generate different views of the project.

This combination is specially important. Much of the data obtained by a tool is complementary to those got by some other. All of them together add up important information about the evolution and behavior of the project, which an unique tool could not capture.

3 Case study: characterization of Mono

Mono [10] is a libre implementation of the .NET Developer Framework [11], lead by a small Boston-based company called Ximian which has about half a dozen full-time or part-time employees assigned to the project. Compared to other 'classical' libre software projects, Mono is a young one, with less than two years of development. We have chosen it to illustrate the results of the methodology because it is a good example of a middle-to-big libre software project, not too big so that it can easily researched in time and scope, not too small so that it is not relevant[6]. Mono is also interesting in itself because it merges some 'classical' software engineering techniques (object orientation, unit testing) with the expertise in libre software that the developers at Ximian acquired during their participation in the GNOME project[7].

Mono's CVS repository contains three development modules: mcs, which contains the hierarchy of classes for the C# language; mono, the C# compiler; and gkt-sharp, a binding of the graphical toolkit Gtk to the C# language. In the following subsections, the results of the methodology on those three modules are presented.

3.1 Commits

Figure 1 shows the evolution of the commits versus. time. It shows clearly how the mcs module is currently the most active, doubling the activity of the other two, but also that in the early stages, mono was developed roughly in parallel with mcs.

To interpret this behavior, one could imagine that at the beginning of the project a lot of effort had to be devoted to create the compiler, certainly the core of the project. Once it had reached a certain maturity, the primary interest shifted towards mcs, which gives value-added functionality to the whole framework. That could also explain why gtk-sharp is younger: essential components have to be developed first, while other modules can be constructed upon them later.

changed, etc. Each version of the committed file can be later retrieved. by the studied project.

[4] All of them are available as libre software (except for those developed by our group, which will be released shortly). This implies that other research groups may easily verify our results as well as audit or enhance the tools.

[5] A physical SLOC is defined as follows: *"a physical source line of code (SLOC) is a line ending in a newline or end-of-file marker, and which contains at least one non-whitespace non-comment character"*.

[6] With over 32,000 commits, it would have been the fourth most active project if hosted at SourceForge [7], by number of commits.

[7] One of the founders of Ximian was Miguel de Icaza, one of the GNOME leaders, and many of its developers are well-known and recognized GNOME hackers

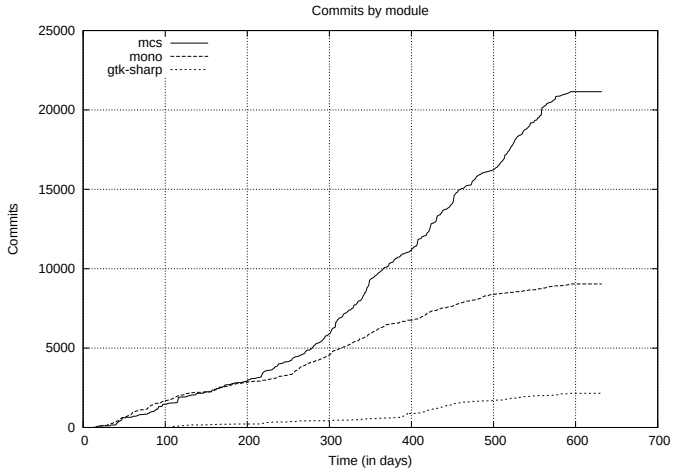

Figure 1. Commits vs. time

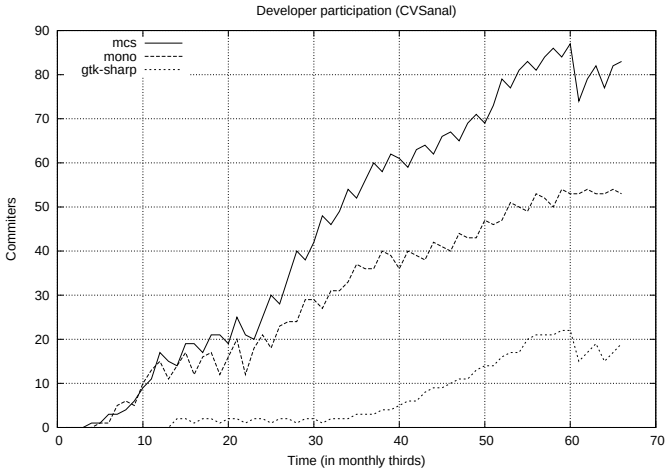

Figure 2. Committers vs. time

module	Number of Committers
mcs	92
mono	65
gtk-sharp	22

Table 1. Committers per module

The commits per day graph (not shown for lack of space) displays an interesting behavior. The peaks of activity for the mcs module usually happen in days close to releases, in particular some days before and after the formal release. The former are probably because authors want to ship some code with the imminent release, while the latter are probably due to fixes originated by bug and error reports.

3.2 Committers

The figures in table 3.2 provide some information about the manpower devoted to the project. In it, a committer is a developers who has done at least one commit for the module.

In figure 2 months have been divided in three thirds of ten days each, and for each third the number of different active committers[8] has been counted. All the lines have a tendency to grow, but that the growth is not monotonic. For instance, the decrease in the 60th third (December 21st to December 31st, 2002) matches Christmas celebration. Mono is a project leaded by a company, but with many volunteer developers. It would be interesting to know whether this kind of behavior is more or less acute when developers are mainly volunteers or when they are mainly employees to work in the project. On the other hand, it is worthwhile to note that the Mono project has a small number of developer withdrawal: in the latest periods over 85% of the committers have contributed with at least one commit.

[8] Active committers are those who have done at least one commit during a give period

3.3 Authorship attribution from source code

Figure 3 keeps analyzing developer information, from a different point of view. The information analyzed now is obtained directly from the source code annotations, not from the CVS logs, which permits the identification of developers without CVS write permission. The first evidence is that the number of committers and authors appears to be similar only for the gtk-sharp module. The number of authors for the mcs module is 75% larger than the number of committers, but it is 50% smaller for mono. The reason for this is unclear, but some comments can be made. The mcs module contains some external code that has been imported into it. Its authors appear only in the headings of the files, but have not committed the code into CVS themselves. In the case of mono, the difference could be attributed to the fact that 'missing' authors sum up altogether less than 100 commits (over a total of over 9000 commits for the whole mono module): their contribution could be too small to get any credits in the source code.

Figures 4 and 5 show the total number of authors found in the mcs and mono modules as well as the number of authors whose contribution is superior to three given percentages. It can be observed how although the number of authors increases over time, the number of developers who have contributed with more than 10%, 5% and 3% of the total code tends to a constant value. It is also shown how while for mcs there are three clear different values for each percentage, in the case of the mono module they all tend to

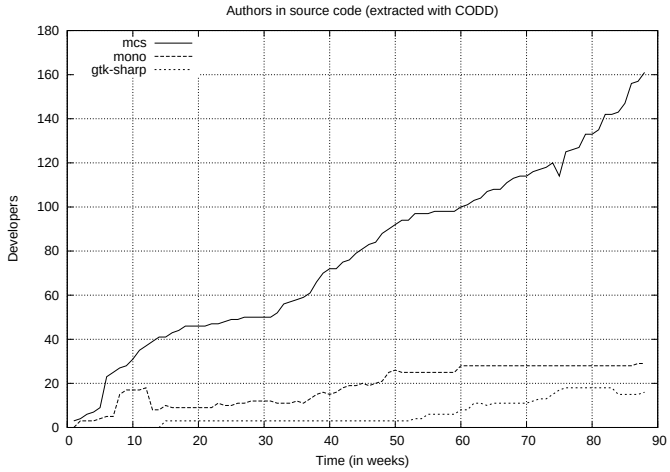

Figure 3. Authors vs. time

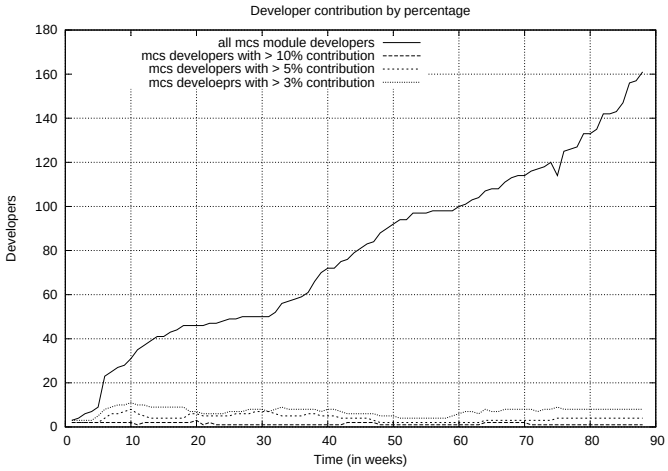

Figure 4. Developers contributing (in mcs) vs. time

the same value. This means that contributions to mcs are more 'staggered' than in the case of mono. This could be a consequence of mono being a project developed primarily by a few 'main' developers. The other contributions done by others are comparatively small.

Figures 4 and 5 show several anomalies. For instance, there is noticeable decrease in mono between the 12th and 13th week. It can be explained by some code been removed from the repository: the libffi library[9], which caused some problems when installing with Cygwin and that was removed once it was not needed anymore (as documented in the mono-list mailing list). Running CODD on the libffi directory the week before the removal shows how this library has been developed primarily by employees of Red Hat, Software AG and HP. Once removed these entries do not appear anymore, causing a decrease in the number of authors found.

3.4 Lines of code

In figure 6 the evolution of the code (in physical lines of code versus time) is shown. The evolution of the several modules studied can be appreciated (with rather different growth rates). Some periods with large restructurations resulting in the dropage of several thousands lines of code are clearly visible (see the line for mcs). The relevance of each module in different stages of the project is also clearly visible.

As a side note, although it was shown how the decrease in authors in mono between the 12th and 13th week is clear, it is hardly noticeable in the SLOC evolution.

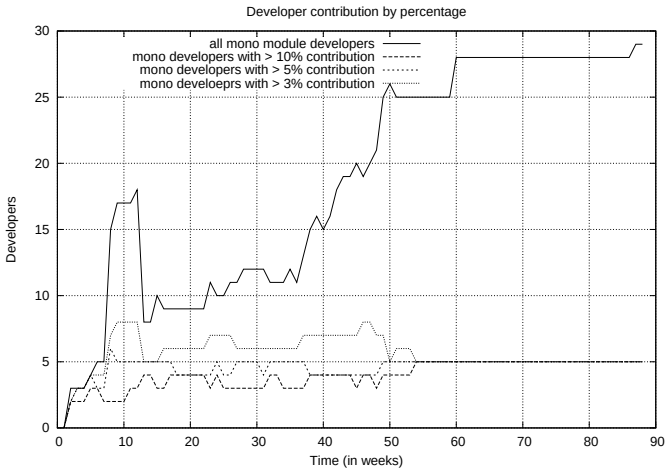

Figure 5. Developers contributing (to mono) vs. time

[9]The libffi library provides a portable, high level programming interface to various calling conventions.

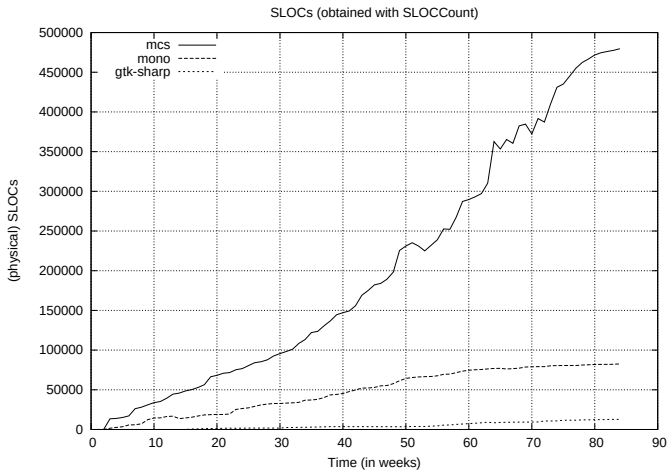

Figure 6. Source lines of code vs. time

4 Conclusions and further work

Libre software development permit analysis of unprecedent depth and detail for a fully reviewable and repeatable software engineering study [8]. The huge amount of information available for lots of libre software projects, with a great variety in size, programming language, programming tools, programming methods, etc. offers the possibility of creating a comparison framework from which knowledge and experience can be gained. The qualitative comparisons that are nowadays usually made could be completed with quantitative data taken at any given point of the life of a project. This allows for the having realistic pictures of the status of a project and its evolution, which should be very valuable to the managers and developers of the project. In addition, data crossing and comparisons between the different sources used in the described methodology gives software engineers a wider perspective of the studied projects.

Many work has still to be done to enhance the tools used to implement the methodology, and the methodology itself, in order to obtain more insight on the process of creating libre software. In this respect, although this paper gives an idea of a methodological approach, it is in fact very limited. Future research should include many other parameters that have not been taken into account yet. For instance, the use of complexity measures (McCabe [9] or Halstead [6]) is one of the considerations that are missing and should be introduced. From other point of view, more specific correlations should be studied so that characterization of a project from some points of view would permit the inference of other information more difficult to obtain.

References

[1] N. Bezroukov. A second look at the cathedral and the bazar. *First Monday*, 1997. http://www.firstmonday.dk/issues/issue4_12/bezroukov/.

[2] Codd. http://codd.berlios.de/.

[3] R. A. Ghosh and V. V. Prakash. The orbiten free software survey, May 2000. http://www.firstmonday.dk/issues/issue5_7/ghosh/.

[4] R. A. Ghosh, G. Robles, and R. Glott. Software source code survey (free/libre and open source software: Survey and study). Technical report, International Institute of Infonomics. University of Maastricht, The Netherlands, June 2002. http://www.infonomics.nl/FLOSS/report.

[5] J. M. González-Barahona, M. A. Ortuño Pérez, P. de las Heras Quirós, J. Centeno González, and V. Matellán Olivera. Counting potatoes: The size of Debian 2.2. *Upgrade Magazine*, II(6):60–66, Dec. 2001. http://people.debian.org/~jgb/debian-counting/counting-potatoes/.

[6] M. H. Halstead. *Elements of Software Science*. Elsevier, New York, USA, 1977.

[7] K. Healy and A. Schussman. The ecology of open-source software development. Technical report, University of Arizona, USA, Jan. 2003. http://opensource.mit.edu/papers/healyschussman.pdf.

[8] S. Koch and G. Schneider. Results from software engineering research into open source development projects using public data. *Diskussionspapiere zum Tätigkeitsfeld Informationsverarbeitung und Informationswirtschaft*, (22), 2000. http://wwwai.wu-wien.ac.at/~koch/forschung/sw-eng/wp22.pdf.

[9] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 1976.

[10] Mono. http://www.go-mono.com/.

[11] .NET developer framework. http://msdn.microsoft.com/library/default.asp?url=/nhp/default.asp?cont%entid=28000519.

[12] E. S. Raymond. The cathedral and the bazar. *First Monday*, 1997. http://www.firstmonday.dk/issues/issue3_3/raymond/.

[13] Sloccount. http://www.dwheeler.com/sloccount/.

[14] D. A. Wheeler. More than a gigabuck: Estimating gnu/linux's size, June 2001. http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html.