# The Challenges of Automated Quantitative Analysis of Open Source Software Projects*

Daniel M. German
Computer Science Department
University of Victoria
dmgerman@uvic.ca

Gregorio Robles, Jesus M. Gonzalez-Barahona
Grupo de Sistemas y Comunicaciones
Universidad Rey Juan Carlos (Madrid, Spain)
{grex,jgb}@gsyc.escet.urjc.es

April 2006

Assessing libre (free/open source) software applications has become a main issue in recent days, with several evaluation frameworks proposed. One of the main goals of all these frameworks is to automate the evaluation process as much as possible, and to provide a universal, comparable metric for the suitability of a libre software application. In this position paper we discuss in detail the drawbacks that in our opinion have current frameworks, based on the experiences gained from researching free/open source software software applications empirically for several years.

**Abstract**

## 1 Introduction

Libre (free/open source) software has gained lately much attention from companies and public administrations, being no more a field almost exclusive of computer enthusiasts and hackers. This situation raises new challenges, as new potential users need to evaluate functionality, quality, technical assistance, and other aspects that determine the how adequate the software is for the intended use. To address some of those challenges, several evaluation frameworks for the use of libre software applications, such as the Business Readiness Rating (OpenBRR) or QSOS, have been proposed recently. This position paper offers a critical review of these evaluation frameworks from a practical point of view, based on the experience of researchers who have been working in

*The work of Gregorio Robles and Jesus M. Gonzalez-Barahona has been funded in part by the European Commission, under the CALIBRE CA, IST program, contract number 004337 and under the FLOSSWorld SA, IST program, contract number 015722.

the field of extracting facts empirically from libre software projects for several years now.

# 2 Major issues

The OpenBRR framework proposes several metrics that are expected to be automatically computed (functionality, usability, quality, security, performance, scalability, architecture, support, documentation, adoption, community and professionalism). These metrics, or the underlying properties of libre software systems which they try to quantify, have been the matter of research in the empirical software engineering field for some time. As a result of this research, there are some lessons that could be of application in the context of OpenBRR and other similar frameworks. Some of them are introduced in the rest of this section, trying to show how obtaining meaningful metrics for libre software projects may be a tricky business.

## 2.1 Universality

One of the goals of the evaluation frameworks in general, and of OpenBRR in particular, is to obtain a final mark (or grade), "universal" in the sense that it may be not only used to compare among applications in the same domain, but also with any other (in other domains). If achieved, this goal could be of course a very good indicator, since applications with which users are experienced may serve as a point of reference for the suitability of other pieces of software. For instance, a user could state that "according to OpenBRR this application is as good as Apache". But comparing software programs in different application domains can be very difficult, and the result comparison be also difficult to interpret. For instance, how can be compared a feature-rich application (such as OpenOffice.org) against a command line tool that does it job and does it well (such as many Unix command line programs, for instance *wc*)?

From the point of view of an end-user, the decision to use an application should be based on, primarily, fitness of purpose: how good does a given application satisfy the requirements of the user. Only when two applications satisfy the minimal requirements of a user, then other factors can be used for their comparison.

An analogy from the sports world can clarify this issue: it is very difficult, if not impossible to state that two teams from different sports arts are equivalent. Have been Michael Jordan's Bulls in the nineties been a better team than Brazil's association football during the seventies? On the other hand one can say Brazil football team was better than the rest of the world football teams in 1970.

Therefore, it seems that metrics should not be absolute, but relative. From the point of view of a business, the decision is usually not to choose between two applications in different domains. On the contrary, the most common problem is to select one between the collection of available applications in the same domain. The idea of considering absolute numbers as the base for a metric (such as the number of messages per day in the user mailing list) becomes more meaningful when it can be compared. This will make it possible asserting facts such as "application X has twice the number of users than application Y in the same domain", which would mean that, other facts being equal, that X should be preferred.

## 2.2 Activity, size, and length are frequently misleading

The automatic measurement of artifacts (code, bug reports, etc.) and actors (developers, users, etc.) related to libre software projects may also be challenging. Some of these are simple to quantify. For example, it is easy to calculate the number of files or lines of code in a software product. But it is more difficult to estimate the number of contributors to the project, or the number of defects that have been detected and reported. To illustrate this fact, some examples are offered below.

For instance, it is not easy to scan mailing lists identifying each contributor precisely (and inferring from it the number of people involved in discussions [3]): some people change mailing list frequently, some mailing lists are fed by programs that inform automatically of certain events, some projects might require (given its nature, or given the stage in which they are) more discussion than others. Even high traffic on a mailing list may not be positive. Another scenario is organization and decision making, which means that, particularly for some nascent projects, discussions in mailing lists are usually composed of a large number of messages. Once a project matures less messages are necessary. The process of a mature project that has gone through these types of discussions has been enhanced, but a metric the-more-messages-the-better would suggest the opposite. Even spam is a problem: some mailing lists use good spam detection software, or strict moderation policies, that filter most of it, while others do absolutely nothing about it.

Counting bugs, which at first consideration may seem simple, given the existence of a defect tracking system in the project, can also be challenging. To begin with, maybe the project does not use such a system (or started to use it late in its life). Policies for merging bug reports, for considering (or not) reports related to non-current versions, or even for using the defect tracking system for other issues (such as feature requests) can also blur the real number of bugs found. Unless the process for defect tracking is almost identical, comparing them can be challenging.

With respect to developers, it is not uncommon to consider the number of developers of a project as a proxy (and even a direct quantification) of its activity. However, counting developers may be not easy [4]. For example, the authors have found projects in which very few developers commit changes to the version control system, while a large community of contributors send "patches" to these developers (for example, in PostgreSQL –the database management system– during 2005 there were 2 persons committing most of the code to the version control system, but there were 40 contributors of patches during that year [1]). Some researchers have also tried to measure the size of this community of contributors by scanning the logs of the changes for certain words such as "patch", "contributed", or an email address. These heuristics sometimes work, but not always, they are not necessarily universal.

Privacy is also becoming more and more of an issue. The email address is now perceived by many as private information, and therefore it is more difficult to identify contributors in logs using their email address (it is easy to scan for email addresses, but difficult to scan for names). For example in PostgreSQL a contributor of a patch is acknowledged by adding his or her name to the commit log, but it does not contain an email address. Many heuristics that have been frequently used to identify patch contributors rely on an email address.

### 2.3 History

Some projects are better than others at keeping records of their development and in making them available to external observers.

For example, some projects switch software development tools, and this switch might impact their metrics. A common case is Subversion replacing CVS as version control system. Some projects take good care of porting the history from CVS to subversion, but others do not, therefore losing important information about their past.

In other cases a project is a "fork" from another one. Should the history of the new project include the older history of the original project too? Should a project that just forked be considered new or as old as its predecessor?

In other cases a project is divided into different subprojects, for instance when a part of a large project is considered to have enough identity by itself, and is spawned. The (new) smaller projects will have little history to analyze, or it should be analyzed in the context of their (large) predecessor.

### 2.4 Different policies, different data to analyze

Although most libre software projects has a lot of information about themselves in their public repositories, not always that information is comparable (or even available) due to different policies. For instance, source code is always (per definition of libre software) accessible for analysis. But not all projects maintain the source code history. And even when this history is kept and available, it can be used with different semantics. For instance, the different policies on comments when committing to a version control system can range from "no policy at all" to a very detailed set of mandatory information (such as ChangeLogs).

Another interesting case is the use of mailing lists. While in some projects all technical discussions, feature requests, and even fixes and patches, have to be posted to mailing lists, in some others they are strictly restricted to private email between the interested parties, and these discussions are never visible to the rest of the project (and hence they are also invisible to outsiders).

Once more, these different policies mean that the information available in the repositories be of a very different nature from project to project, of course producing different results if it is analyzed with the same methods, and not having into account these differences.

## 3 Conclusions

Measuring without understanding the software process of an application can be risky [2]. Differences from project to project are too large for simple numeric comparisons to be of interest in most cases. Although software metrics are a promising methodology for comparing the performance and other aspects of projects, simplistic quantification may not be of interest to assess the suitability of an application for a given user.

Of course, this does not mean that frameworks such as OpenBRR are not of interest. On the contrary, they are a good starting point, if their outcome is correctly understood, and their limitations are clearly outlined. With this position paper, the authors have tried to point out some of these limitations. Further work should be performed in order to study how to overcome these limitations, so that these evaluation frameworks can be improved.

# References


[1] D. M. German. A study of the contributors of PostgreSQL. In *3rd International Workshop on Mining Software Repositories–MSR Challenge Reports (MSR 2006)*, May 2005. To be presented.

[2] Daniel M. Germán. Using software trails to reconstruct the evolution of software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6):367–384, 2004.

[3] Gregorio Robles and Jesús M. González-Barahona. Developer identification methods for integrated data from various sources. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 106–110, St. Louis, Missouri, USA, May 2005.

[4] Gregorio Robles, Jesús M. González-Barahona, José Centeno-González, Vicente Matellán-Olivera, and Luis Rodero-Merino. Studying the evolution of libre software projects using publicly available data. In *Proceedings of the 3rd Workshop on Open Source Software Engineering*, pages 111–115, Portland, Oregon, USA, 2003.