IGI PUBLISHING **ITJ 4505**

701 E. Chocolate Avenue, Suite 200, Hershey PA 17033-1240, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.igi-global.com

This paper appears in the publication, International Journal of Open Source Software & Processes, Volume 1, Issue 1
edited by **Stefan Koch © 2008, IGI Global**

Tools for the Study of the Usual Data Sources found in Libre Software Projects

Gregorio Robles, Universidad Rey Juan Carlos, Spain

Jesús M. González-Barahona, Universidad Rey Juan Carlos, Spain

Daniel Izquierdo-Cortazar, Universidad Rey Juan Carlos, Spain

Israel Herraiz, Universidad Rey Juan Carlos, Spain

ABSTRACT

Due to the open nature of Free/Libre/Open Source software projects, researchers have gained access to a rich set of development-related information. Although this information is publicly available on the Internet, obtaining and analyzing it in a convenient way is not an easy task and many considerations have to be taken into account. In this paper we present the most important data sources that can be found in libre software projects and that are studied by the research community: source code, source code management systems, mailing lists and bug tracking systems. We will give advice for the problems that can be found when retrieving and preparing the data sources for a posterior analysis, as well as provide information about the tools that support these tasks.

Keywords: bug tracking; mailing list; open source software; software metrics; software repository mining; source code management

INTRODUCTION

The fact that communication and organization are heavily tied in libre software[1] projects to the use of telematic means and that these interactions are, in general, stored and publicly offered over the Internet makes the number of data sources where development information can be extracted from grow beyond source code. In addition, the ability of having memory (as data from activities in the past can be obtained) offers the possibility of performing longitudinal analysis as well. Research groups from all around the world have already taken benefit from the availability of such a rich amount of data sources in the last years. Nonetheless, the access, retrieval and fact extraction is by no means a simple task and many considerations

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

have to be considered to successfully mine the data sources.

This article is probably the first attempt to have a detailed description of the most common data sources that can generally be found for libre software projects on the Internet and the data that can be found in them. In addition, we present some available tools that might help researchers obtaining and partially analyzing the described data sources. These data sources comprise **source code**, **source code management** (in the following, SCM), **mailing lists archives**, and **bug tracking system** (in the following, BTS).

Mining and analyzing these data sources offer an ample amount of possibilities that surpass or complement other data-acquiring methodologies such as surveys, interviews or experiments. The amount of data that can be obtained, in a detailed way and in many cases for the whole lifetime of a software project, gives a precise description of the history of a project (Bauer and Pizka, 2003). In this sense, we have access to the activities (the what), the points in time (the when), the actors (the who) and sometimes even the reason (the why) (Hahsler and Koch, 2005). Compared to surveys, mining these data sources allows to access data for thousands of developers and a wide range of software projects. Most of these efforts can be considered as non-intrusive, as researchers can analyze the projects without interacting with developers. But even in a small environment, e.g., when evaluating the impact of software tools in a small team (Atkins et al., 2002), the use of data from one or more of these sources provides additional insight. Furthermore, mining software repositories has many advantages compared to conducting experiments as real-world software projects are taken into consideration (Mockus and Votta, 2000, Graves and Mockus, 1998).

The structure of this article is as follows: the next section handles the identification of the data sources as well as its retrieval process. Next, various analysis on source code are introduced (hierarchy, file discrimination, analysis of *traditional* source code files, analysis of the rest of files (such as documentation, multimedia, etc.), and authorship). The fourth section presents how SCM systems can be mined, putting special attention on the CVSAnalY tool and some details to be considered when performing analyses on CVS. The fifth section presents the most common format in which mailing lists are stored (MBOX), while the sixth one is devoted to present the data to be found in a BTS. Finally, the reader can find a short summary of the article in the last section.

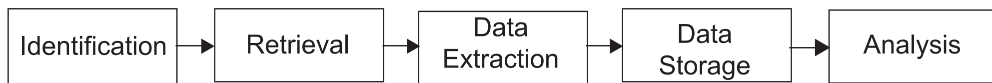
IDENTIFICATION OF DATA SOURCES AND RETRIEVAL

There are some steps before the analysis of data from libre software projects can be started that should be considered: identification and retrieval. It should be noted that there may be several ways of accessing the data, depending on the projects. This is because of the use of the several development-supporting tools that projects use and of having different usage conventions (for instance, the use of tags, comments, among others, may differ from one project to another). The complexity and feasibility of both activities depend on the data source and on the project. Figure 1 gives a diagram that shows the steps that have to be accomplished for any source considered in our study.

In general terms, the identification of the data source depends mostly on its significance for the software development of a project. Hence, identifying the source code, the SCM system, the mailing lists or the BTS is in no way problematic as it lies in the interest of the projects that feedback is provided by users in an easy and fast way. In these cases, the biggest drawback is the lack of historical data. Sometimes we only have a partial set of the data, and in the worst cases nothing at all. This situation is common for software releases, where finding historical versions of the software is sometimes not possible. Other situations where this might happen is when a development tool has not been used in the early stages of development. This is the case of many projects that start using a

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 1. Whole process: from identification of the data sources to analysis of the data



SCM system once the project has gained certain momentum. Having only partial data can also be the result of a migration from one tool to another, losing in the way some information if not all. When researching libre software projects, these considerations have to be taken into account.

But there exist other data sources for libre software projects that are not so obvious and hence their identification is not that straightforward. For instance, organizational information that is embedded into some format and that is beyond the use of standard tools as SCM systems, mailing lists and BTS. In general, such type of information is project-dependent and can be only obtained for one project or a small number of them. This is the case for packaging systems such as the .deb format used in Debian and Debian-based distributions or the .rpm Red Hat package system in use in Red Hat and other distributions. But beyond this, we can find project-related information in other places such as the Debian Popularity Contest (Robles et al., 2006b) or the Debian Developer database (Robles et al., 2005). Other data sources may also be considered; for instance, in KDE there is a file that is used to list all the ones who have write access to their SCM repository. Another example is given in a study by Tuomi (Tuomi, 2004) in which the credits file, a text file listing all important contributors to the project, of the Linux kernel are studied in detail. Identification of the data source requires in such cases specific knowledge on the project and is difficult if not impossible to be generalized.

Once the data source has been identified, it has to be retrieved to a local machine in order to be analyzed (see Figure 1). Although this process may not seem to be very difficult at first, previous experiences have shown that some considerations and good practices should be followed in this step as reported by Howison et al. in the retrieval of information from the web pages hosted at SourceForge (Howison and Crowston, 2004). For instance, the analysis of the credits file, which can be found together with the sources in many projects, has to deal with the complexity that there is no standardized way of naming the authors, so projects follow their own conventions.

In the next sections we will enter into detail in the process of data extraction and data storage once the data have been properly retrieved from the information source to a local machine.

SOURCE CODE

We should begin with the concept of release. It is important due to the fact that it points out the main milestone happened during the life of a project. It usually has a common nomenclature which is akin to “MM.mm.bb”. Where “MM” means the number of the major release, “mm” means the number of minor releases and “bb” connotes some bug fixes and small improvements.

As software development projects, source code is the central point of all interactions, being a primary way of communication and playing a major signaling and coordination role. According to (Lanzara and Morner, 2003), source code “is transient knowledge: it reflects what has been programmed and developed up to that point, resuming past development and knowledge and pointing to future experiments and future knowledge.”.

The study of the source code, as the main product of the software development process, is a matter that has been done for over thirty years now. But not only *traditional* source code

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

(i.e., programmed in a programming language) can be taken into account, but also all the other elements that make the software, such as documentation, translation, user interface and other files (Robles et al., 2006a).

The analysis usually starts with a source code base that is stored in a directory (or alternatively in a compressed directory, usually in tar.gz or tar.bz2 format common in the libre software world). After decompressing the tarball, if needed, the hierarchical structure of the source code tree is identified and stored.

Then, files can be grouped into several categories depending on type (as will be described below) which allows for a more specific analysis. This means, for instance that source code files in a programming language can be analyzed differently than images or documentation files. On the other hand, the discrimination for files with source code can be finer, identifying the programming language and offering the possibility of using alternative metrics depending on it. As a consequence, object oriented metrics could be applied to files containing Java code, but would not be required for files that are written in assembler language.

The whole process can be observed in Figure 2: after (possibly) decompressing, the directory and file hierarchy is obtained, then files are discriminated by their type and finally analyzed, if possible taking into consideration the file type that has been identified in the previous step. In the following subsections the different steps are described more in detail.

Hierarchical Structure

The structure of directories and files of a software program (and how it changes over time) has already been the focus of some research studies (Capiluppi, 2004, Capiluppi et al., 2004). The idea is that the technical architecture and probably therefore the organization of the development team is mapped by the tree hierarchy of directories. So, from a directory hierarchy, we could infer the organizational structure of a libre software development project.

File Discrimination

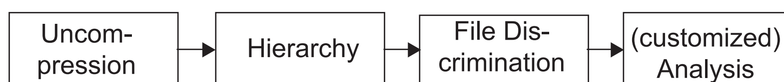
File discrimination is a technique that is used to specifically analyze files on behalf of their content (Robles et al., 2006a). The most common way of discriminating files is by using heuristics, which may vary in their accuracy as well as in the granularity of their results.

A first set of heuristics may determine the type of a file by considering its extension. File extensions are non-mandatory, but usually conventions exist so that the identification of the content of a file can be made easier and to enable the automation of administrative tasks.

Hence, a first step for file discrimination consists of having a list of extensions that links to the content of the file. In this context, the *.pl* extension is indicative for a file that contains programming instructions while a *.png* can be considered as an image file. Of course, this can be done at several granularity levels, meaning that a *.c* file is a file that with high probability contains programming language, being that the programming language C code. Table 1 shows an excerpt of the list of file extensions that can be used.

The file types that can be considered are documentation, images, internationalization (i18n) and localization (l10n), user interface (ui), multimedia and code files. For the latter type, a more detailed analysis and discrimination between source code that is part of the software

Figure 2. Process of source code analysis



Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

application (code) from the one that helps in the building process (generally Makefiles, configure.in, among others) and from documentation files that are tightly bound to the development and building process (such as README, TODO or HACKING) can be made.

A second step in the process of file discrimination includes inspection of the content of the files both to check if the identification made by means of matching file extensions is correct and to identify files that have no extension or whose extension is not included in the previous list.

In this case, heuristics are generally content-specific and may go more in depth depending on the detail of discrimination we are looking for. One of the most common ways to improve file discrimination by looking at the file content is to analyze the first line. There exists some convention in source code files that denotes that the programming language that they contain. For instance, in the case of a file written in the Python, *Bourne again* shell or Perl programming language (examples can be found in Table 2), the first line could contain respectively the following information[2].

In the case of programming languages, further information can be gained from the structure of the code, by the identification of specific keywords or other elements such as specific comments. For text files (especially the ones that are based on mark-up languages), tags and other specific elements may help in the identification process. Finally, other algorithms can be taken into account, as the information returned by the UNIX *file* command on the file type (which also identifies some of the binary formats, especially useful in the case of images).

Some of the previous discrimination techniques are already in use in some tools, most notably in SLOCCount (see (Wheeler, 2001, Robles et al., 2006b)). As SLOCCount counts the number of lines of code it is only concerned with identifying source code files and identifying the programming language in which they are written, not considering all other file types that we have taken into consideration in this work (documentation, translations, and other).

Table 1. (Incomplete) set of matches performed to identify the different file types

File type	Extension/file name matching
documentation	*.html *.txt *.ps *.tex *.sgml
images	*.png *.jpg *.jpeg *.bmp *.gif
i18n	*.po *.pot *.mo *.charset
ui	*.desktop *.ui *.xpm *.theme
multimedia	*.mp3 *.ogg *.wav. *.au *.mid
code	*.c *.h *.cc *.pl *.java *.s *.ada
build	configure.* makefile.* *.make
devel-doc	readme* changelog* todo* hacking*

Analysis of Source Code Files

The analysis of source code files is one of the most known tasks. There exist an ample number of measures that can be and have been extracted directly from the source code, among other its length (in lines of code or source lines of code), complexity measures (as the popular ones proposed by Halstead (Halstead, 1977) and McCabe (McCabe, 1976)) or even composite metrics such as the Maintainability Index (Oman and Hagemeister, 1992).

Table 2. Examples of first line indicating that the file is written in Python, Shell or Perl respectively

#! /usr/bin/python
#! /usr/bin/sh
#! /usr/bin/perl

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

The availability of a certain range of tools for this purpose makes the conception of a tool that integrates all of them a primary task. The goals of the integration is to make it possible to extract all the metrics and facts from source code files by using several tools in a simple and most uniform way. The tools used to measure the code should be, if possible, used as *black boxes*, so that the integration tool does not need to know or adapt its inner functioning. In addition, the integration tool should handle the input to and the output from the measurement tools to ease its use.

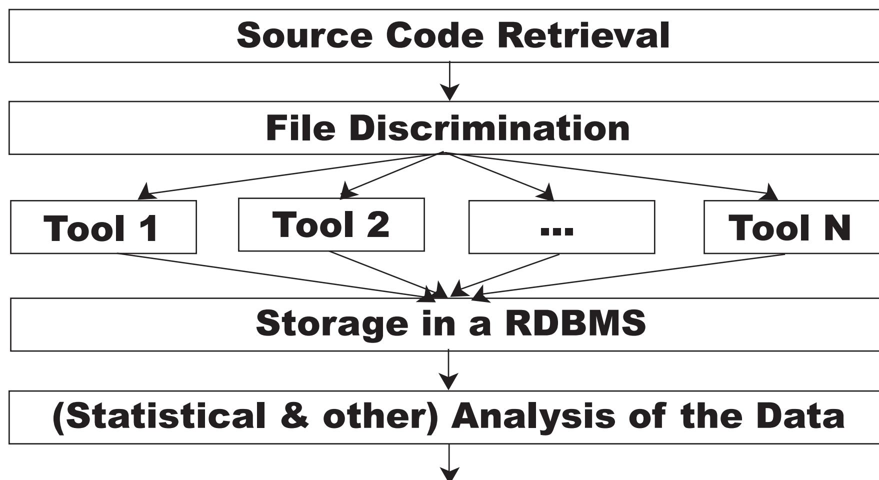
That is precisely what can be done with GlueTheos[3], a tool designed and implemented by the authors of this article: a system with an architecture that allows the data retrieval and analysis of public software development data repositories. The structure of the GlueTheos tool is presented in Figure 3, and consists of a module for downloading (if required, with a periodical pattern) the sources to be analyzed, to examine the content of the sources on a file basis, to run the tools depending on the file type, to identify the results and store them properly in a relational database system and finally to provide results.

The current version can access *CVS*, *Subversion* and *git*. File discrimination allows to run the tools specifically on the files where this makes sense. Hence, if we had a tool that returns object oriented measures from Java files it would make no sense to run it on a shell script. This step then optimizes the analysis to be performed.

The next step is the heart of GlueTheos and consists of running the different tools on the source code and retrieving the data that these tools return. GlueTheos has been designed in a way in which it does not require to adapt the tools it integrates, hence facing the complexity of the various ways of calling them and the various ways of obtaining their results. Both calling and returning have been solved following an object-oriented approach, so that for any tool only the differences have to be implemented.

The calling procedure requires information such as the way a tool has to be called (mainly the path to the executable), the input that the

Figure 3. Architecture of the GlueTheos tool



Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

tool requires (usually a file or a directory) and the type of output that the tool returns (again, usually a file or a directory).

The returning methods depend on the type of output that the analysis tools provide. If it is a file, the number of returning elements has to be given and the special character that is used to separate them (usually a tabulator, a white space or a comma). In general, the path that gives the filename of the file that has been analyzed is also returned, so its position has to be specified.

After retrieving and storing the data from external tools, GlueTheos has to consider only the data in the database to obtain statistical and other results from the data set. This includes some procedures to enhance the database structure in order to normalize the fields or to obtain intermediate tables with statistical information that is of common use.

Analysis of Other Files

Besides source code written in a programming language, we identify other artifacts that compose the sources of libre software projects. (Robles et al., 2006a) shows the many possibilities that arise from the study of those files, but other references to this issue may be found in related literature. Some authors have focused on the analysis of the change log files (Capiluppi et al., 2003) as they usually follow a common pattern in libre software projects, although sometimes this pattern is slightly different from the standardized way used in GNU projects[4].

Translation files may be used to keep track of the amount of translation work that has been accomplished to the moment and hence have a quantitative manner of knowing the support of that software in a given language.

Regarding licenses, in addition of a reference to the licensing terms that can be found at the top of the code files, usually projects have a text file which includes the full text of the license. The filename may give enough evidence for the type of license that a project uses, but other ways can also be considered. One that we have been trying with is the use of a locality-sensitive hash like nilsimsa (Chang and Mockus, 2008). This type of hashes return codes with small changes for inputs that differ only slightly. As intellectual property issues have become a recent area of interest among industry, some approaches (and tools, such as FOSSology) have been presented that target these problems (Gobeille, 2008).

Finally, the amount of documentation for a software system could be a good topic for empirical research. In this sense, the *doceval*[5] tool offers a way of assessing and partially evaluating the documentation that can be found in the sources of libre software projects (Robles et al., 2006c).

Authorship Analysis

Usually, source code files contain copyright and license information in their first lines (Spinellis, 2003). So, for instance, the notice in the apps/units.c file of the GIMP project shown in Table 3 clearly states that the copyright holders are Spencer Kimball and Peter Mattis and that the license in use is the GNU General Public License.

Table 3. Excerpt of a copyright statement found in the GIMP project

```
/* The GIMP – an image manipulation program
 * Copyright (C) 1995 Spencer Kimball and Peter Mattis
 *
 * This program is free software; you can redistribute it and/or modify
[...]
```

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

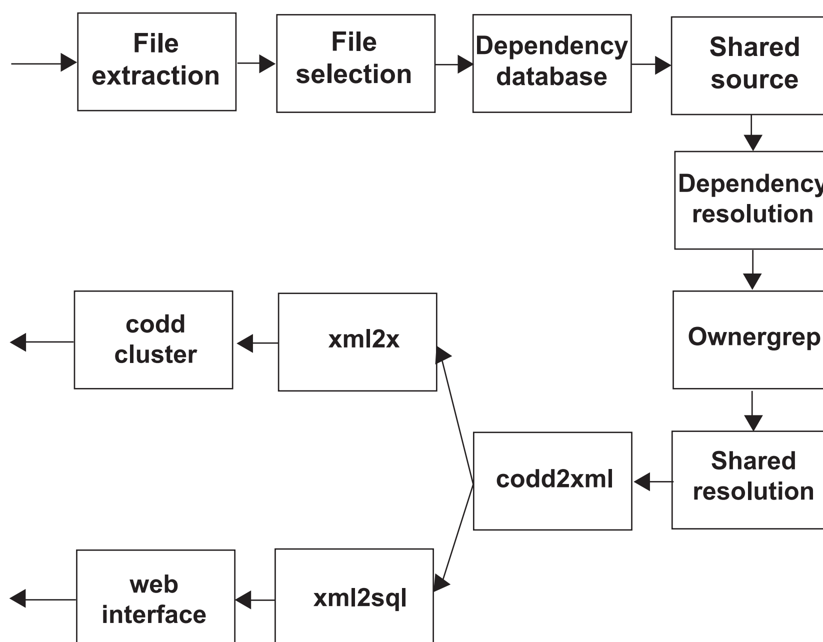
CODD[6] is a tool that searches for authorship information in source code by tracking copyright notices and other information in the headings of files (Ghosh and Prakash, 2000, Ghosh et al., 2002). It assigns the length (in bytes) of each file to the corresponding authors. The process that CODD follows to obtain these results are shown in Figure 4.

File extraction is composed of the *init* subroutine which takes the source code package (or packages) that are given through the command line by the user, decompresses them if necessary, and tries to identify recursively the files that the package contains.

During the file selection all source code files, documentation, interfaces and not-resolved implementations are taken together with their size in bytes, their MD5 sum and their relative route in the package and stored in a *codd*[7]. Files are selected by means of their extension, so for instance the *.c* file extension is categorized into source code files (usually they correspond to C files). CODD stores the *.h* files that have a *.c* in the same package as interfaces (the algorithm that is used here depends partially on the programming language that is being analyzed). Calls to an interface in source code files (for instance *.c* files for C) that do not have their corresponding interface in the same package (a *.h* for C) will be classified in the non-resolved implementations category, that in a future step will be handled for dependency resolution.

In a third step two databases are created in order to find shared source code and dependencies. In the first one, named *codefile_signatures*, all the MD5 sums of the files are stored. The second one contains all the interfaces that were found in the previous step. MD5 is a type of hash that allows to know if two files are equal; if they are they will have the same MD5 hash value. MD5 hashes are very interesting when the source code file is exactly the same, but a single modification (i.e., when it is committed

Figure 4. Process of the CODD tool



Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

into the SCM of the new project the RCT-type id changes) makes it impossible to recognize it as a shared file.

In order to find shared source code, CODD runs another time through all codds and looks if the source code files appear more than once in the database (really it looks if the name and MD5 sum appear more than once). If this occurs, the file is located in at least two different packages. A similar process is used to resolve dependencies. CODD will search for not-resolved implementations in the codds and compare their MD5 sums with the ones that are stored in the interfaces database. A list with all the packages where this interface is implemented will be inserted as well.

The owner grep block is the one that is responsible for looking for authorship contributions. It runs again through all source code and documentation files and scans authorship attribution by means of certain heuristics. Mainly the heuristics look for several patterns: email addresses [a], copyright notices [b] and software control versioning ids [c]. Information about the authors is stored in the *credits* section of the codds. The regular expressions that have been used are following:

[a] Email grep: [\d\w_\=\.\%]+?\@[\d\w\._\-]+?\.\w+?)(?=[\s:>\n\r\)]|$

[b] Copyright grep: .*copyright (?:\(c\))?[\d\,\-\s\:]+(?:by\s+)?([?\d]*)

[c] Id grep: (?:Id|Header).*?\d\d\:\d\d\:\d\d (\S+?) \S+?

Next, the resolution of shared source code is done. In the *shared* source code section of the codds we still have files and a list of packages that contain these files. As these files can only be assigned to a single package (in order to avoid double counting the contribution of an author), CODD looks for its author (running again the *ownergrep* algorithm) in the file and assigns it to the package in which the author is the main contributor.

The last blocks of Figure 4 show that the codds can be then transformed to an intermediate and independent format (as for instance XML and SQL).

CODD is a very powerful tool, but it has some weaknesses. The most important one is that it lacks a way of merging the various ways in which an author may appear. So, authors may appear several times with different names or e-mail addresses. For instance, we have found that some developers have up to 15 e-mail addresses. In the case of companies, the same may happen; so, IBM or the MIT appear in several ways (up to ten times!) with slightly different wordings (Robles et al., 2007).

Cleaning of the data should also be enhanced. The heuristics that are used in CODD have proved to be very powerful, but cannot avoid that developers use different conventions to assign copyright. Most of these problems could be solved by a set of more powerful heuristics.

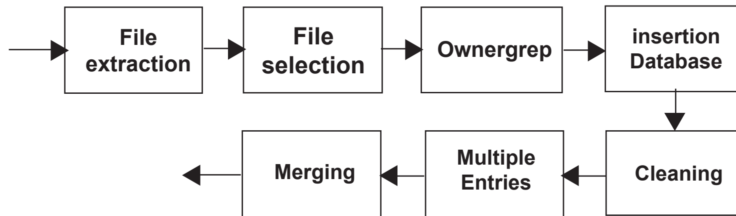
As *CODD* raises some limitations regarding authorship identification, the authors decided 2004 to create a new tool from scratch based on the heuristics given by CODD. This tool has been called *pyTernity*[8]. The architecture of pyTernity is identical to the one described for CODD as it can be seen from Figure 5, although it lacks of all the procedures for identifying dependencies among files.

The most innovative elements are the ones that consider data cleaning and the identification of multiple entries. For the former, entries in database are removed from elements that make them different; this goes from additional white-spaces to the avoidance of dots. Some heuristics have been set up for this, although they have been complemented with a database of frequent changes. Cleaning includes splitting up an entry when it is due to two or more authors. So, the entry "Spencer Kimball and Peter Mattis" will result in two, one for Spencer Kimball and another one for Peter Mattis. If this is the case, both names appear as authors of the file and get attributed half of its length (in bytes or lines of code).

The latter part comprises the identification of multiple entries. Developers may appear in

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 5. Process of the pyTernity tool



several ways, making results very unsatisfactory. The first efforts in this sense went into the construction of a large database where the various entries identified for a given developer were merged into a unique one. This has proved to enhance results in a prominent way. Other, more complex, routines may be used for extracting names from e-mails, with procedures from the machine learning world as for instance applying *named entity recognition* (Minkov et al., 2005).

Once cleaning has been performed and multiple entries have been identified, pyTernity merges the entries in the database so that authors appear only once in a file. This procedure implies to add all the contributions by an author, so it adds the lengths of each entry (in bytes or lines of code).

SCM SYSTEM META-DATA

Generally speaking, most libre software projects use a SCM system to manage file versions during the development process. They allow to track changes and past states of a software project. Thus, obtaining the current and any past state of the code is made possible by the use of a SCM system. This allows to make source code analyses as we have presented them in the previous section in a longitudinal manner and to extract facts on the evolution of a software project.

But beyond this, SCM systems store a set of meta-data of the changes. These meta-data can be tracked and analyzed. This information is usually related to the interactions that occur among developers and the SCM systems. In general the information is only related to actions that comprehend write access while reading (downloading the sources) or obtaining other information (diffs, among others) cannot be tracked in that way. For instance, along with a change, valuable information is recorded, like the date of change, the full path where the change occurred, user who committed or the comment written by the committer[9].

Here, we present a tool that analyzes the interactions that occur between developers and the most used SCM systems used in libre software projects at the current time, CVS, Subversion, *git* and *Bazaar*. This tool, which has been labeled CVSAnalY, is based on the analysis of the SCM system log entries and implements all the theoretical details that will be presented in this section (Robles et al., 2004). Another tool, called SoftChange, has been used for similar purposes by German *et al*. (Germán and Hindle, 2005).

In CVSAnalY any interaction -also called commit- a committer does with the central SCM system repository is logged with following data associated (some aforementioned): committer name, date, file, revision number, lines added, lines removed and an explanatory comment introduced by the committer. There is some file-specific information that can also be extracted, as for instance if the file has been removed[10]. On the other hand, the human-inserted comment can also be parsed in order to see if the commit corresponds to an external contribution or even to an automated script.

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Basically CVSAnalY consists of three main steps, preprocessing, insertion into database and post-processing, but they can be subdivided into several more as it has been done in Figure 6. In the following subsections the inner functioning of CVSAnalY will be presented, focusing on details of its use with CVS. Its use with other SCM systems should be similar.

Preprocessing: Retrieval and Parsing

Preprocessing includes downloading the sources from the repository of the project in study. Once this is done, the logs are retrieved and parsed to transform the information contained in log format into a more structured format (SQL for databases or XML for data exchange).

Besides the information for every commit, other data obtained from the parsing requires some attention. Although committers seldom change their username, sometimes this happens, so the various usernames have to be merged into a unique one. For instance, in the KDE project committers usually get an account prior to a *kde.org* e-mail address. If a developer is afterwards assigned an e-mail address the username of e-mail and SCM system have to be identical for organizational and practical reasons. If the username in the e-mail address is different from the CVS username, CVSAnalY syncs with the former one and the actions done with both usernames are considered as done by a unique developer.

The following is a CVS log excerpt for the AUTHORS file of the KDevelop project[11]. It gives the last three revisions (from revision 1.47 to 1.49) done during the last months of the year 2003 until mid-2004. Log messages from other SCM systems, such as Subversion, *git* or *Bazaar* look similar.

```
[...]
RCS file: /mirrors/kde//kdevelop/AUTHORS,v
Working file: /mirrors/kde//kdevelop/AUTHORS
head: 1.49
branch:
locks: strict
access list:
keyword substitution: kv
total revisions: 103;   selected revisions: 103
description:
----------------------------
revision 1.49
date: 2004/06/21 18:57:13; author: rgruber; state: Exp; lines: +4 -0
Added self
----------------------------
revision 1.48
date: 2004/02/24 14:42:59; author: dagerbo; state: Exp; lines: +5 -1
Added self :)
----------------------------
revision 1.47
date: 2004/02/15 22:40:33; author: aclu; state: Exp; lines: +3 -3
Some more credits update.
[...]
```

While being parsed each file is also matched for its type. Usually this is done by looking at its extension, although other common filenames (for instance README or TODO) are also looked for. The goal of this separation is to identify different contributor groups that work on the software, so besides source code files the following file types are also considered: documentation (including web pages), images, translation (generally internationalization and localization), user interface and sound files.

Figure 6. Process of the CVSAnalY tool

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Files that do not match any extension or particular filename are accounted as unknown. This discrimination follows the criteria that have been presented in section 3.2, although it lacks the possibility of looking at the content of the files as we only consider filenames (because this is the only information that appears in the CVS logs).

CVS also has some peculiarities when introducing contents for the first time (this action is called initial check-in). The initial version (with version number 1.1.1.1) is not considered in our computation as it is the same as the second one (which has version number 1.1). The number of aggregated and removed lines in CVS are computed from this initial version. This means that the first commit (the initial check-in) logs as if 0 lines were added. This does not correspond to reality. In order to obtain the actual number of LOCs in the first version we count the LOCs by means of the UNIX *wc* tool[12] of the latest version, subtracting the added lines and adding the removed lines of all the other commits.

Comments attached to commits are usually forwarded to a mailing list so that developers keep track of the latest changes in CVS. Some projects have established some conventions so that certain commits do not produce a message to the mailing list. This happens when those commits are supposed to not require any notification to the rest of the development team. A good example of the pertinent use of *silent* commits comes from the existence of bots that do several tasks automatically.

In any case, such conventions are not limited to non-human bots, as human committers usually may also use them. In a large community -as it is the case for the ones we are researching- we can argue that *silent* commits can be considered as not contributory (i.e., changes to the head of the files, for instance a change in the license or the year in the copyright notice, or moving many files from one location to another). Therefore, we have set a flag for such commits in order to compute them separately or leave them out completely in our analysis.

For instance, the developers of the KDE project mark such commits with the comment *CVS_SILENT* as it can be seen from following log excerpt extracted from the kdevelop_scripting.desktop file of the KDevelop CVS module. In this case it is due to a change to a *desktop* file, a file type that is related to the user interface. Being this change not considered interesting for other developers to know about, the author of this commit decided to make this commit *silently*.

```
[...]
RCS file: /mirrors/kde//kdevelop/kdevelop_scripting.desktop,v
Working file: /mirrors/kde//kdevelop/kdevelop_scripting.desktop
head: 1.24
branch:
locks: strict
access list:
keyword substitution: kv
total revisions: 30;    selected revisions: 30
description:
----------------------------
revision 1.24
date: 2005/03/28 03:29:25; author: scripty; state: Exp; lines: +2 -2
CVS_SILENT made messages (.desktop file)
----------------------------
[...]
```

Write access to the SCM system is not given to anyone. Usually this privilege is granted only to those contributors who have reached a compromise with the project and the project's goals. But external contributions -commonly called patches, that may contain bug fixes as well as implementation of new functionality- from people outside the ones who have write access (committers) are always welcome.

It is a widely accepted practice to mark an external contribution with an authorship attribution when committing it. Thus, we have constructed certain heuristics to find and mark commits due to such contributions. The heuristics we have set up are based on the appearance of two circumstances: patch (or patches in its plural form) together with a preposition (from,

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

by, of, and other) or an e-mail address or an indication that the code had been attached to a bug fix in the BTS. The regular expressions that have been used are following:

[a] patch(es)?\s?.* from [f] patch(es)?\s?.* by
[b] patch(es)?\s.*@ [g] @.* patch(es)?
[c] ?s.* patch(es)? [h] s? .* patch(es)?
[d] patch(es)? of [i] <.* [Aa][Tt] .*>
[e] attached to #

As an example, the following slightly modified excerpt taken from the kdevelop.m4.in file from the KDevelop module in the KDE CVS repository shows a patch applied by a committer with username "dymo" that was submitted originally by Willem Boschman:

[...]
----
revision 1.39
date: 2004/06/11 17:07:57; author: dymo; state: Exp; lines: +3 -3
Applied patch from Willem Boschman -
fix builddir != srcdir configuration problem.

[...]

All these efforts have in common that they perform text-based analysis of the comments attached by committers to the changes they perform. The range of possibilities in this sense is very ample. For instance, Mockus *et al.*, and later on in an enhanced manner Amor *et al.*, have tried to identify the reasons for changes (classifying changes as adaptative, perfective or corrective) in the software using text-analysis techniques (Mockus and Votta, 2000, Amor et al., 2006).

Data Treatment and Storage

Once the logs have been parsed and transformed into a more structured format, some summarizing and database optimization information is computed and data is stored into a database.

Usually the output of the previous parsing consists of a single database table with an entry per commit. This means that information is stored in a raw form, the table containing possibly millions of entries depending on the size and age of a project. Information is hence in a raw format and in an inconvenient way if we consider getting statistical information for developers and projects from it.

A first step in this direction is to make use of normalization techniques for the data. In this sense, committers are assigned a unique numerical identification and if further granularity is needed, procedures have been implemented to do the same at the directory and file level. For the sake of optimization this has been introduced during the parsing phase so additional queries do not have to be performed. The next step is to gather statistical information on both committers and modules. These additional tables will give detail on the interactions by contributors or to modules, which is one of the most frequent information that is asked.

Additional information that makes longitudinal analyses possible is the evolution of contributions by developers and to modules. Hence, the same statistical queries that have been obtained for committers and modules for the summarizing tables can be obtained in a monthly or weekly basis since the date the repository was set up.

On the other hand, unfortunately CVS does not keep track of which files have been committed at the same time. The absence of this concept in CVS may bring some distortion into our analysis. We have therefore implemented the sliding window algorithm proposed by German (Germán, 2004) and Zimmermann et al. (Zimmermann et al., 2005) that identifies atomic commits (also known as *modification requests* or transactions) by grouping commits from the CVS logs that have been done (almost) simultaneously. This algorithm considers that commits performed by the same committer in a given time interval (usually in the range of seconds to minutes) can be considered as an atomic commit. If the time window is fixed, the amount of time that is considered from the first commit to the last one is a constant value. For a sliding time window, the time interval is not

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

constant; the time window is restarted for every new commit that belongs to the same transaction until no new commit occurs in the (new) time slot (Zimmermann et al., 2005).

The post-process is composed of several scripts that interact with the database, statistically analyze its information, compute several inequality and concentration indexes and generate graphs for the evolution over time for a couple of interesting parameters (commits, committers, LOCs...). Results are shown through a publicly accessible web interface that allows easy inspection of the whole repository (general results), a single module or by committers[13]. Therefore results are available for remote analysis and interpretation by project participants and other stakeholders.

MAILING LISTS ARCHIVES (AND FORUMS)

Mailing lists and forums are the key elements for information dissemination and project organization in libre software projects. Without almost any exception, libre software projects provide one or more mailing lists. Depending on the project, many mailing lists may exist for several target audiences. So, for instance, SourceForge recommends to open three mailing lists: a technical one for developers, another one to give support to users and and a third one that is used for announcing new releases.

Mailing lists are programs that forward e-mail messages they receive to a list of subscribed e-mail addresses. More sophisticated mailing list managers have plenty of functionality which allows for easy subscription, unsubscription, storage of the messages that have been sent (known as the archives), and avoidance of spam, among others.

Forums are web-based programs that allow visitors to interact in a similar manner as in an e-mail thread with the difference that in this case all the process goes through HTML forms and that results are visible on the web.

Both mailing lists and forums are based on similar concepts and their differences lie in their implementation and the need for different clients to participate in them. Mailing lists require the use of an e-mail client, while forums can be accessed through web browsers. As their concept is the same, there exist some software programs that transform mailing lists messages to a forum-like interface and vice-versa. Because of that, in this article we will only focus on mailing lists, specifically on one of the most used mailing lists managers called GNU Mailman[14] and the RFC 822 (also known as MBOX) format in which it generally stores and publishes the archives.

The RFC 822 Standard Format

As mentioned above, generally all mailing list managers offer the possibility of storing all posts (the archives) and making them publicly available through a web interfaces. This offers the possibility for newcomers to go through the history and to gain knowledge on technical as well as organizational details of a project.

The archives are also offered in text files following the MBOX format. MBOX is a format used traditionally in UNIX environments for the local storage of e-mail messages. It is a plain text file that contains an arbitrary number of messages. Each message is composed of a special line followed by an e-mail message in the RFC-822 standard format. The special line that allows to differentiate messages consists of the keyword “From” followed by a blank space, the poster’s e-mail address, another blank space and finally the date the message was sent. The RFC-822 format can be divided into two parts: (a) headers, that contain information for the delivery of the message and (b) the content, which is the information to be delivered to the receiver; the standard only allows lines of text, so filtering has to be implemented if an image or other information is attached.

Mailing lists in MBOX format can be analyzed by means of the MailingListStats, or *mlstats* for short, tool[15]. Given an URL of the archives of the mailing lists, *mlstats* outputs the information extracted from the headers and the

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

content of the message in database format for further processing and analysis.

Below is an excerpt of a post sent to a mailing list that has been stored following the RFC-822 standard. Is is an automatic message sent April 30 2005 to the GNOME CVS mailing list. This list keeps track of all the commits that are done to the CVS system of the GNOME project. This assures that subscribers are aware of the latest changes in the CVS. The content of the message, the description of the modification that had been performed, has been omitted in the excerpt.

From gnomecvs@container.gnome.org Sat Apr 30 20:16:38 2005
Return-Path: <gnomecvs@container.gnome.org>
X-Original-To: cvs-commits-list@mail.gnome.org
Delivered-To: cvs-commits-list@mail.gnome.org
To: cvs-commits-list@mail.gnome.org
X-CVS-Module: marlin
Message-Id: <20050501001636.0C5E-A165E4A@container.gnome.org>
Date: Sat, 30 Apr 2005 20:16:36 -0400 (EDT)
From: gnomecvs@container.gnome.org (Gnome CVS User)
X-Virus-Scanned: by amavisd-new at gnome.org
Cc:
Subject: GNOME CVS: marlin iain
X-BeenThere: cvs-commits-list@gnome.org
X-Mailman-Version: 2.1.5
Precedence: list
Reply-To: gnome-hackers@gnome.org
List-Id: CVS Logs <cvs-commits-list.gnome.org>
List-Unsubscribe: <http://mail.gnome.org/mailman/listinfo/cvs-commits-list>,
<mailto:cvs-commits-list-request@gnome.org?subject=unsubscribe>
List-Archive: </archives>
List-Post: <mailto:cvs-commits-list@gnome.org>
List-Help: <mailto:cvs-commits-list-request@gnome.org?subject=help>
List-Subscribe: <http://mail.gnome.org/mailman/listinfo/cvs-commits-list>,
<mailto:cvs-commits-list-request@gnome.org?subject=subscribe>
X-List-Received-Date: Sun, 01 May 2005 00:16:38 -0000
[Here comes the body of the post which has been omitted in this excerpt]

From the message excerpt above, we can see some of the headers that are described in the standard. The most important ones are following: *From* (e-mail address, sometimes also real name, of the sender), *Sender* (address of the responsible entity for the last transmission), *Reply-To* (address the author wants to be replied), *To* (address(es) of the receiver(s)), *Cc* (e-mail address(es) of the receiver(s) that should receive a copy), *Bcc* (addressee(s) with *carbon* copy), *Subject* (usually contains a brief description of the topic), *Received* (contains address of the intermediate machine that has transferred the message), *Date* (when the message was sent given by the sender machine), *Message-ID* (unique identifier of this message), *In-reply-to* (Identifier of the parent message to which the current one is a response), and *References* (identifications (message-IDs) of all the other messages that are part of the conversation thread).

In addition to the data that can be found in the headers, some other information could be obtained from analyzing the content of the messages. In this regard, Weißgerber *et al*. analyze the type of patches first sent to mailing lists and later on integrated into the source code tree of the project (Weißgerber et al., 2008).

BUG-TRACKING SYSTEMS

BTS are used in libre software projects to manage the incoming error and feature enhancement reports from users and co-developers. The use of BTS is relatively extended and the most

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

known tool in this area is BugZilla[16], a BTS developed by the Mozilla project that has been adopted by other large projects as well. Hence BugZilla is the system we study in this article, although conceptually all other systems should work similarly.

BugZilla allows to manage all bug reports and feature requests by means of a publicly available web interface. Besides the reports, it also offers the possibility of adding comments so that developers may ask for further information about the error or other end-users may comment it. Beyond BugZilla, other tools exist with similar features, as for instance GNATS (the one used in the FreeBSD project). SourceForge and other web platforms that support software development have implemented their own BTS for the projects they host.

Data Description

BugZilla stores in its database specific information for each bug report. The fields that can be usually found are following[17]:

- **Bugid:** Unique identifier for any bug report.
- **Description:** Textual description of the error report.
- **Opened:** Date the report was sent.
- **Status:** Status of the report. It can take one of the following status: new, assigned (to a developer to fix it), reopened (when it has been wrongly labeled as resolved), needinfo (developers require more information), verified, closed, resolved and unconfirmed.
- **Resolution:** Action to be performed on the bug. It can take following status: obsolete (will not be fixed as it is a bug to a previous, already solved issue), invalid (not a valid bug), incomplete (the bug has not been completely fixed), notgnome (the bug is not of GNOME, but of a component of another project, as for instance X window system or the Linux kernel), notabug (the issue is not really a bug), wontfix (the developers consider not to correct this error for any reason) and fixed (the error has been corrected).
- **Assigned:** Name and/or e-mail address of the developer in charge of fixing this bug.
- **Priority:** Urgency of the error. It can take following values: immediate, urgent, high, normal and low. Usually this field is modified by the bugmaster as users do not have sufficient knowledge on the software to know the correct value.
- **Severity:** How this error affects the use and development of the software. Possible values are (from high severity to lower one): blocker, critical, major, normal, minor, trivial and enhancement.
- **Reporter:** Name and e-mail address of the bug reporter.
- **Product:** Software that contains the bug. Usually this is given at the tarball level.
- **Version:** Version number of the product. If no version was introduced, *unspecified* is given. Also, for enhancements the option *unversioned enhancement* may be chosen.
- **Component:** Minor component of the product.
- **Platform:** Operating system or architecture where the error appeared.

Usually all fields (besides the automatic ones like *bugid*, the opening date or its status) are filled out the first time by the reporter. Larger projects usually have some professional or volunteer staff that review the entries in order to adjust the information (Villa, 2003, Villa, 2005). This is especially important for fields like priority or severity as end-users hardly have no knowledge or experience on how to evaluate these fields.

Data Acquisition and Further Processing

For the analysis of the data stored in a BTS, we have created a preliminary tool that is specifically devoted to extract data from BugZilla. The architecture of the BugZilla Analyzing Tool is

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

described in Figure 7. Although the retrieval of the data could theoretically be simplified by obtaining the database of the BugZilla system from the project administrators, we thought that retrieving the data directly from the web interface would be more in accordance with the non-intrusive policy that all other tools described in this article follow.

We had to deal with several problems while retrieving the BugZilla data. After crawling for all web pages (one per bug) and storing them locally, we had to transform the HTML data into an intermediate log-type format, as not all fields were given for all bugs due probably to a transition from a previous system. Probably also because of this, there may have been some information loss and some ids could not be tracked. Other problems that we found, were the existence of wrong date entries for some bugs and comments. As the bug report ids are sequential, we could fix these entries when we found out that the date was wrong. We applied the same solution to comments with erroneous dates, as comments are also posted sequentially and cannot be introduced before the bug report has been submitted.

In recent versions of BugZilla, it is possible to obtain the data in XML format which simplifies in a great manner the data extraction[18]. When writing this article, the use of the XML interface was not as common as the author would wish, so retrieving the data from parsing web pages was the unique non-intrusive manner at that time. In any case, the BugZilla analyzing tool has been designed in such a way that only by removing some parts (specifically the specific HTML-parser which parses into the independent format) and by modifying the generic parser we could reuse the rest of the modules without major changes using the XML query format. This is also valid for other BTS, as GNATS.

One of the issues of BTS is that in general the most relevant information in a bug report is included in natural language (usually in English) in the *Description* field . Bettenburg *et al*. have proposed a tool that extracts structural information such as source code (i.e., patches), listings, etc. from it (Bettenburg et al., 2008).

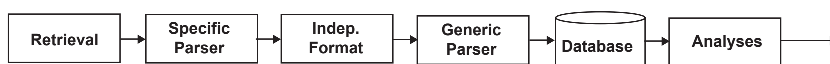
SUMMARY

Libre software projects offer a vast amount of information about their development process and the resulting product. Although this information is publicly available over the Internet, researchers should take into consideration the many *hidden* problems that may occur when obtaining and properly analyzing these data. In this article we have given some insight into the most data sources in research, its problems, how to circumvent them and, if possible, have provided and introduced tools that may help when researchers mine them.

Table 4 summarizes the data sources described in this article and the tools that can help researchers in their analysis. Regarding source code, there exist an ample amount of tools that have been used for years in corporate software engineering environments. The ones presented here have a specific target on libre software, as they address issues such as licensing of the files (FOSSology), the identification of the authors – or copyright holders – (CODD and pyTernity) or the presence of absence of documentation (*doceval*).

SCM systems are widely used in libre software projects and provide much information about the software development. The analysis of the logs of these systems, by means of tools like CVSAnalY, gives insight on the dynamics of the projects. The combination of source code and

Figure 7. Architecture of the BugZilla analyzer



Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

information from the SCM offers a wide range of analysis, especially concerning patterns over time. In this sense, GlueTheos and SoftChange, retrieve the sources for various points in time and extract some metrics. The final result allows to analyse how software projects have evolved over time in respect to measurements related to source code (growth, complexity, etc.), human resources (number of developers, inequality of contributions, etc.) and activity (number of commits, number of patches, etc.).

Finally, mailing lists and forums are usually the main communication channels used in libre software projects. In this article, we have discussed a tool, mlstats, that analyzes the information that can be found in the header of the mail messages.

Despite the possibilities that the vast amount of publicly available information from libre software projects offer, there are a number of problems, threats and challenges that researchers have to consider when using these data for their activities.

The first major problem comes from incomplete data sets. The cause for this is due to the fact that projects may have switched development-supporting systems and have not migrated old contents into the new system.

A threat to the use of SCM data comes from the necessity of having an account to perform changes. The original developers are in these scenarios not the ones who commit the code into the repository (committers). The validity of some research may be affected by the policies of projects, as for example the inequality of contributions may be artificially skewed towards those who have permission to do the changes in the SCM.

There is a big challenge in merging information from various sources. For instance, correlating bugs in the BTS to commits in SCM and to code in the source code is a tricky task that requires complex methods. In addition, these methods may have to be changed from project to project as the dynamics may be different among them (i.e., in some projects, committers indicate the bug report number in the commit message, while in others patches are not handled via a BTS but through a mailing list)

Finally, as mining libre software projects has become popular among scientists, many projects have suffered from an overflow of data gathering petitions, both automatically by means of tools or directly from humans in the sense of invitations to participate in surveys. In the specific case of tools, sometimes retrieving data has caused the slow down, or denial of service, of servers where the infrastructure of the project is installed, resulting in the tool being banned.

Some of the aforementioned issues are being addressed by the FLOSSMetrics[19] and the FLOSSMole (Conklin et al., 2005) projects, that have as objective to construct, publish and analyse a large scale database with information and metrics about libre software development

Table 4. Summary of data sources, tools and purpose

Data Source	Tool	Purpose
Source Code	FOSSology	Licensing
Source Code	CODD/pyTernity	Authorship (copyright hlder) analysis
Source Code	doceval	Assessment and partial evaluation of documentation
SCM	CVSAnalY	SCM log messages analysis (evolutionary analysis)
Source Code & SCM	GlueTheos	Evolutionary analysis
Source Code & SCM	SoftChange	Evolutionary analysis
Mailing lists & forums	MailingListStat	Analysis of MBOX headers

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

coming from several thousands of software projects. If such initiatives succeed, researchers wanting to study the libre software phenomenon will have an ample amount of data ready to be analyzed, avoiding many of the tasks (identifications, acquisition, extraction and storage) and the threats discussed in this article.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their helpful comments and suggestions. This work has been funded in part by the European Commission, under the FLOSSMETRICS (FP6-IST-5-033547), QUALOSS (FP6-IST-5-033547) and QUALIPSO (FP6-IST-034763) projects, and by the Spanish CICyT, project SobreSalto (TIN2007-66172) and by SEDEPECA.

REFERENCES

Amor, J. J., Robles, G., and González-Barahona, J. M. (2006). Discriminating development activities in versioning systems: A case study. In *Proceedings PROMISE 2006: 2nd. International Workshop on Predictor Models*.

Atkins, D. L., Ball, T., Graves, T. L., and Mockus, A. (2002). Using version control data to evaluate the impact of software tools: A case study of the version editor. *IEEE Transactions on Software Engineering*, 28(7):625–637.

Bauer, A. and Pizka, M. (2003). The contribution of free software to software evolution. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE)*, Helsinki, Finland. IEEE Computer Society.

Bettenburg, N., Premraj, R., Zimmermann, T., and Kim, S. (2008). Extracting structural information from bug reports. In *MSR '08: Proceedings of the 2005 Working Conference on Mining software repositories*.

Capiluppi, A. (2004). Improving comprehension and cooperation through code structure. In *Proceedings of the 4th Workshop on Open Source Software Engineering, 26th International Conference on Software Engineering*, Edinburg, Scotland, UK.

Capiluppi, A., Lago, P., and Morisio, M. (2003). Evidences in the evolution of OS projects through changelog analyses. In *Proceedings of the 3rd International Workshop on Open Source Software Engineering*, Orlando, Florida, USA.

Capiluppi, A., Morisio, M., and Ramil, J. F. (2004). Structural evolution of an Open Source system: a case study. In *Proceedings of the 12th International Workshop on Program Comprehension*, pages 172–183, Bari, Italy.

Chang, H.-F. and Mockus, A. (2008). Evaluation of source code copy detection methods on FreeBSD. In *MSR '08: Proceedings of the 5th Working Conference on Mining software repositories*.

Conklin, M., Howison, J., and Crowston, K. (2005). Collaboration using OSSmole: A repository of FLOSS data and analyses. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 126–130, St. Louis, Missouri, USA.

Germán, D. M. (2004). Mining CVS repositories, the softChange experience. In *Proceedings of the International Workshop on Mining Software Repositories*, Edinburgh, UK.

Germán, D. M. and Hindle, A. (2005). Visualizing the evolution of software using softChange. *Journal of Software Engineering Knowledge Engineering*. Accepted for publication, under revisions.

Ghosh, R. A. and Prakash, V. V. (2000). The orbiten free software survey. *First Monday*, 5(7).

Ghosh, R. A., Robles, G., and Glott, R. (2002). Software source code survey (free/libre and open source software: Survey and study). Technical report, International Institute of Infonomics. University of Maastricht, The Netherlands.

Gobeille, R. (2008). The fossology project. In *MSR '08: Proceedings of the 5th Working Conference on Mining software repositories*.

Graves, T. L. and Mockus, A. (1998). Inferring change effort from configuration management databases. In *5th IEEE International Software Metrics Symposium*, pages 267–, Bethesda, Maryland, USA.

Hahsler, M. and Koch, S. (2005). Discussion of a large-scale open source data collection methodology. In *Proceedings of the Hawaii International Conference on System Sciences (HICSS-38)*, Big Island, Hawaii, USA.

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Halstead, M. H. (1977). *Elements of Software Science*. Elsevier, New York, USA.

Howison, J. and Crowston, K. (2004). The perils and pitfalls of mining SourceForge. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 7–11, Edinburg, Scotland, UK.

Lanzara, G. F. and Morner, M. (2003). The knowledge ecology of open-source software projects. In *Proceedings of the 19th EGOS (European Group of Organizational Studies) Colloquim*.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320.

Minkov, E., Wang, R., and Cohen, W. (2005). Extracting personal names from emails: Applying named entity recognition to informal text. In *Proceedings of the Human Language Technology Conference. Conference on Empirical Methods in Natural Language Processing*, Vancouver, B.C., Canada.

Mockus, A. and Votta, L. G. (2000). Identifying reasons for software changes using historic databases. In *Proc Intl Conf Softw Maintenance*, pages 120–130.

Oman, P. and Hagemeister, J. (1992). Metrics for assessing a software system's maintainability. In *International Conference on Software Maintenance*, pages 337–344, Los Alamitos, CA. IEEE Computer Society Press.

Robles, G., Dueñas, S., and González-Barahona, J. M. (2007). Corporate involvement of libre software: Study of presence in debian code over time. In *OSS*, pages 121–132.

Robles, G., González-Barahona, J. M., and Guervós, J. J. M. (2006a). Beyond source code: The importance of other artifacts in software development (a case study). *Journal of Systems and Software*, 79(9):1233–1248.

Robles, G., Gonzalez-Barahona, J. M., Michlmayr, M., and Amor, J. J. (2006b). Mining large software compilations over time: Another perspective of software evolution. In *Third International Workshop on Mining Software Repositories*, pages 3–9, Shanghai, China.

Robles, G., González-Barahona, J. M., and Michlmayr, M. (2005). Evolution of volunteer participation in libre software projects: evidence from Debian. In *1st International Conference on Open Source Systems*, pages 100–107, Genoa, Italy.

Robles, G., Koch, S., and González-Barahona, J. M. (2004). Remote analysis and measurement of libre software systems by means of the CVSAnalY tool. In *Proc 2nd Workshop on Remote Analysis and Measurement of Software Systems*, pages 51–56, Edinburg, UK.

Robles, G., Prieto-Martínez, J. L., and González-Barahona, J. M. (2006c). Assessing and evaluating documentation in libre software projects. In *Proceedings of the Workshop on Evaluation Frameworks for Open Source Software (EFOSS 2006)*.

Spinellis, D. (2003). *Code Reading: The Open Source Perspective*. Addison Wesley Professional.

Tuomi, I. (2004). Evolution of the Linux Credits file: Methodological challenges and reference data for Open Source research. *First Monday*, 9(6). http://www.firstmonday.dk/issues/issue9_6/ghosh/

Villa, L. (2003). How gnome learned to stop worrying and love the bug. In *Otawa Linux Symposium*, Otawa.

Villa, L. (2005). Why everyone needs a bugmaster. In *linux.conf.au*, Canberra.

Weißgerber, P., Neu, D., and Diehl, S. (2008). Small patches get in! In *MSR '08: Proceedings of the 2005 Working Conference on Mining software repositories*.

Wheeler, D. A. (2001). More than a gigabuck: Estimating GNU/Linux's size. http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html.

Zimmermann, T., Weißgerber, P., Diehl, S., and Zeller, A. (2005). Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445.

ENDNOTES

[1] In this article the term "libre software" is used to refer to any software licensed under terms that are compliant with the definition of "free software" by the Free Software Foundation, and the definition of "open source software" by the Open Source Initiative, thus avoiding the controversy between those two terms.

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

[2] The location of the binaries may depend from system to system, although the standard location for them is the /usr/bin directory.

[3] GlueTheos is named after its purpose to glue different tools together in an easy way. Hence, this program is the *god*, *theos* in Greek, of gluing some already existing tools together. It can be retrieved from http://tools.libresoft.es/gluetheos.

[4] In the GNU coding standards, some conventions for change log files are given, see http://www.gnu.org/prep/standards/html_node/Change-Logs.html

[5] *doceval* can be obtained from https://forja.rediris.es/projects/csl-doceval/.

[6] The most current version of CODD may be found at http://libresoft.es/Tools/CODD.

[7] CODD uses as intermediate storage a file for each source package which are called the *codd* files.

[8] The most current version of pyTernity may be found at http://tools.libresoft.es/pyternity.

[9] A committer is a person who has write access to the repository and does a commit -an interaction- with it at a given time.

[10] In a SCM system there is actually no file deletion. In the case of CVS, files that are not required anymore are stored in the *Attic* and may be called back anytime in future.

[11] KDevelop is an IDE (Integrated Development Environment) for KDE. More information can be obtained from http://kdevelop.org/.

[12] *wc* is a standard UNIX tool to count lines of files, among others.

[13] See http://libresoft.es/Results

[14] The MailMan's project web site can be found at following URL: http://www.gnu.org/software/mailman/.

[15] http://forge.morfeo-project.org/frs/? group_id=33

[16] http://www.bugzilla.org/

[17] The ones shown next are the ones that can be found for the GNOME BugZilla system. BugZilla can be adapted and modified, so the fields may (and will) change from project to project.

[18] For instance, bug #55,000 from the KDE BTS, which can be accessed through the web interface at http://bugs.kde.org/show_bug.cgi? id=55000 may also be obtained in XML at following URL: http://bugs.kde.org/xml.cgi? id=55000.

[19] http://flossmetrics.org

Gregorio Robles is associate professor at the Universidad Rey Juan Carlos in Móstoles, Spain. He earned a degree on electrical engineering from the Universidad Politécnica de Madrid (studying his last year and submitting his master thesis at the Technical University of Berlin) and obtained his PhD in 2006. His research work is centered in the study of libre software development from an engineering point of view and especially with regard to quantitative and empirical issues. Other, non-technical related matters have also been of his interest like volunteer-driven software development and social network analysis in the libre software phenomenon. He has developed or collaborated in the design of programmes to analyse libre software and the tools used to produce them. He was also involved in the FLOSS study on libre software financed by the European Commission IST programme and in other European-funded projects such as FLOSSMetrics or QualOSS.

Jesús M. González-Barahona teaches and researches in Universidad Rey Juan Carlos, Móstoles (Spain). He started to be involved in the promotion of libre software in 1991. Since then, he has carried on several activities in this area, including the organization of seminars and courses, and the participation in working groups on libre software, both at the Spanish and European levels. Currently he collaborates with several

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

libre software projects (including Debian) and associations, writes in several media about topics related to libre software, and consults for companies and public administrations on issues related to their strategy on these topics. His research interests include understanding libre software development, where he has published several papers, and is participating in some international research projects. He is also one of the promoters of the idea of an European master program on libre software, and has specific interest in the education in that area.

Daniel Izquierdo-Cortazar is a PhD student at the Universidad Rey Juan Carlos in Móstoles, Spain. He earned a degree in computer science from the same university and obtained his master's degree in computer networks and computer science systems in 2006. His research work is centered in the assesment of libre software communities from an engineering point of view and especially with regard to quantitative and empirical issues. Right now he holds a grant from the Universidad Rey Juan Carlos to dedicate part of his time to his PhD thesis. He is also involved in European-funded projects such as QualOSS or FLOSS-World. He has also had the opportunity of attending to Wirtschaftsuniversität Wien (3 months in 2007) as a research visitor.

Israel Herraiz is a PhD student at the Universidad Rey Juan Carlos in Móstoles, Spain. Israel Herraiz holds a bachelor's degree in chemical engineering and master's degree in chemical and mechanical engineering from University of Cadiz (Spain). Right now he holds a grant from the Government of Madrid, to dedicate his full time to his PhD thesis, whose main topic is "Software Evolution of Large Libre Software Projects". In particular, he is using time series analysis and other statistical methods to characterize the evolution of software projects. He has participated in several research projects funded by the Framework Programme of the European Commision such as QualOSS or CALIBRE. He has also collaborated on other projects funded by companies such as Vodafone and Telefonica. He has participated in the writing of manuals about managing and starting libre software projects.

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.