Mining Large Software Compilations over Time: Another Perspective of Software Evolution*

Gregorio Robles, Jesus M. Gonzalez-Barahona
Universidad Rey Juan Carlos
{grex,jgb}@gsyc.escet.urjc.es

Martin Michlmayr
University of Cambridge
martin@michlmayr.org

Juan Jose Amor
Universidad Rey Juan Carlos
jjamor@gsyc.escet.urjc.es

ABSTRACT

With the success of libre (free, open source) software, a new type of software compilation has become increasingly common. Such compilations, often referred to as 'distributions', group hundreds, if not thousands, of software applications and libraries written by independent parties into an integrated system. Software compilations raise a number of questions that have not been targeted so far by software evolution, which usually focuses on the evolution of single applications. Undoubtedly, the challenges that software compilations face differ from those found in single software applications. Nevertheless, it can be assumed that both, the evolution of applications and that of software compilations, have similarities and dependencies.

In this sense, we identify a dichotomy, common to that in economics, of software evolution in the small (micro-evolution) and in the large (macro-evolution). The goal of this paper is to study the evolution of a large software compilation, mining the publicly available repository of a well-known Linux distribution, Debian. We will therefore investigate changes related to hundreds of millions of lines of code over seven years. The aspects that will be covered in this paper are size (in terms of number of packages and of number of lines of code), use of programming languages, maintenance of packages and file sizes.

Categories and Subject Descriptors

D.2.m [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Measurement, languages

Keywords

Mining software repositories, large software collections, software evolution, software integrators

1. INTRODUCTION

Large systems based on libre software[1] are developed in a manner that is quite different to traditional systems. In traditional large systems, such as operating systems, most work is done in-house, with only few pieces licensed from other sources and little work contracted to other companies. Such work is also performed in close cooperation with the organization and under tightly defined requirements. Libre software, on the other hand, is typically written by small, independent teams of volunteers, sometimes collaborating with paid staff from one or more companies. While various projects interact with each other, in particular where dependencies between the software exist, there is no central coordination between the individual projects. The main task of vendors (i.e. distributions) of libre operating systems is therefore not to write software but to group existing software, taken from several sources, together and to make that collection easy to install, configure and administer.

Since users of libre software have no incentive to download software from hundreds of sites and installing them individually, distributions play an important role by providing an integrated system that is easy to install. Unsurprisingly, a number of companies have seen this as a business opportunity and offer such distributions among with related services, such as support. There are also a number of community projects which operate on a non-profit basis like other libre software projects. Given their open way of collaboration, these are a good target for in-depth study of extremely large software compilations. While some commercial entities have recently started their own community projects in addition to their enterprise offerings, most notably Fedora (Red Hat) and OpenSUSE (Novell), we will take Debian as the source

*The work of Gregorio Robles, Jesus M. Gonzalez-Barahona and Juan Jose Amor has been funded in part by the European Commission under the CALIBRE CA, IST program, contract number 004337. The work of Martin Michlmayr has been funded in part by Google, Intel and the EPSRC. We would also like to thank the anonymous reviewers for their extensive comments.

MSR'06, May 22–23, 2006, Shanghai, China.

[1] Through this paper we will use the term "libre software" to refer to any code that conforms either to the definition of "free software" (according to the Free Software Foundation) or "open source software" (according to the Open Source Initiative).

of data for this study since it is one of the most accessible and best established projects.

Debian is a community effort that has provided a software distribution based on the Linux kernel for well over 10 years. The work of the members of the Debian project is similar to that carried out in other distributions: software integration. Unlike many other distributions, Debian is mostly composed of volunteers who are spread all around the world. As a side-effect of this, all development infrastructure, including mailing lists, bug tracking and of course the source code itself, is publicly available. In addition to integrating and maintaining software packages, members of the Debian project are in charge of the maintenance of a number of services, such as a web site, user support, etc. In the following, we will mostly focus on the work carried out in their role as integrators of software – work that has had tremendous success, given that Debian is the largest distribution of all in terms of number of software packages [1].

2. RELATED RESEARCH AND GOALS

Software evolution has been a matter of study for more than thirty years now [5, 7]. So far, the scope of software evolution analyses has always been that of single applications. Example case studies are the "classical" analysis of the OS/360 operating system [5], and, more recently, many of studies on libre software systems. Such is the case for the Linux kernel [3], or other well-known libre software applications, including Apache and GCC [11]. Noteworthy is the proposal of studying the evolution of applications at the subsystem level [2], as this introduces the issue of granularity. Nonetheless, our approach considers as system the whole software compilation and as subsystem the hundreds of applications and libraries that are usually matter of software evolution studies.

However, the authors have not found a study on the evolution of a system integrating many independent software applications. Actually, software compilations have rarely been studied in software engineering. This is probably due to the intrinsic difficulties that software companies find when integrating large amounts of software programs built by several vendors. There are a number of reasons for this, both legal and technical. It seems that even if one of the most promising steps of software engineering has been to create reusable components (or modules), in a similar way as bricks and mortar, little attention has been put on how the integration of these components evolve. A promising path has been the study of integration of COTS from a software evolution perspective [6].

As noted above, the public availability of source code of libre software programs and the possibility of freely redistributing this software allow to have an ample number of software distributions. Both characteristics also enable the investigation of distributions. In this sense, there have been already some *radiographies* of some distributions, mainly of the well-known Red Hat and Debian distributions. These studies have pointed out the packages they contain, the size of the packages and of the whole distribution, and some statistics on the programming languages, among other issues [14, 4, 1].

This paper goes a step beyond the *single-version* analyses of software distributions: our goal is to study the evolution of software compilations. We therefore consider data from several points in time. However, it should be noted that the goals of this study differ slightly from those usually considered as common for software evolution. In part this is because a different type of work has to be accomplished while creating software compilations than during software development. The work to be done for a software compilation is mainly integration of software rather than development, although the latter is not excluded at all (for instance, for the development of an installer or other software administration tasks that distributions may include). Needless to say, there are some aspects that are common to *traditional* software evolution analyses, such as how the size of the software evolves.

Putting a software distribution together is not only integration work, however. Maintenance also has to be performed, but not so much in the *classical* way as defined by Swanson (corrective, adaptive and perfective maintenance activities) [12]. Maintenance in software compilations focuses on the integration of new versions of software that has been released. In other words, a package maintainer will not necessarily submit patches that correct errors; but they will update the package whenever new versions are published by the developers of the application or when changes in the distribution, such as library transitions or toolchain updates, occur. This raises interesting questions in our longitudinal analysis. For instance, we will analyze packages that are kept and that get *lost* (removed) over time, as the composition of the software compilation may vary. We will also look at packages whose version has not changed, as we will take this as an indication of *unmaintained* packages.

As software compilations are composed of a large variety of software applications for different purposes and from different backgrounds, we may find a larger heterogeneity than when looking at specific software applications. This is the case for instance in the use of programming languages: a particular software application, for example the Linux kernel [3, 10], is usually implemented primarily in one programming language, with only minor portions in other languages (such as glue code or the build system). This means that studying compilations as large as the one we have selected as our case study can be considered as a proxy of libre software in general – a macroscopic view of the libre software landscape. We are in this sense performing a holistic study of libre software and analyze how it is *in the large*, drawing some conclusions about the phenomenon itself.

3. METHODOLOGY

The methodology that we have used for the analysis of the stable versions of Debian is as follows: first, we have retrieved files which contain information about the packages that are distributed in a given Debian distribution. Distributions are organized internally in packages where packages correspond to applications or libraries. Debian developers commonly try to modularize packages to the maximum, for example splitting documentation into a separate packages if it is very large. Since 2.0, the Debian repository contains a Sources.gz file for each release, listing information about every source package. For each package, it contains the name and version, list of binary packages built from it, name and e-mail address of the maintainer, and some other information that is not relevant for this study. In some cases, packages are not maintained by individual volunteers, but by teams.

As an example, an excerpt of the entry for the Mozilla source package in Debian 2.2 has been included below[2]. It can be seen how it corresponds to version M18-3, provides four binary packages, and is maintained by Frank Belew.

```
[...]
Package: mozilla
Binary: mozilla, mozilla-dev, libnspr4, libnspr4-dev
Version: M18-3
Priority: optional
Section: web
Maintainer: Frank Belew (Myth) <frb@debian.org>
Architecture: any
Directory: dists/potato/main/source/web
Files:
  57ee230[...]c66908a 719 mozilla_M18-3.dsc
  5329346[...]bad03c8 28642415 mozilla_M18.orig.tar.gz
  3adf83d[...]ca20372 18277 mozilla_M18-3.diff.gz
[...]
```

The Sources.gz files are parsed and the data they contain is stored into a database. Then, each package is retrieved to a local machine, the number of source lines of code (SLOC) is counted and the programming languages in which the code is written are recognized. The counting is made by means of SLOCCount[3], a tool written by David Wheeler that gives the number of physical source lines of code of a software program. SLOCCount takes as input a directory where the sources are stored, identifies (by a series of heuristics) the files that contain source code, recognizes for each of them (also by means of heuristics) the programming language, and finally counts the number of source lines of code they contain. SLOCs are parsed differently for different languages, which forces the identification of programming languages.

SLOCCount also identifies identical files (by using MD5 hashes), and includes heuristics to detect (and avoid counting) automatically generated code. These mechanisms are helpful when analyzing the code, but have some deficiencies. Finding almost identical files using such hashes is not very effective. In the second case, heuristics only take care of well-known and/or common cases, but do not detect all of them, or others that may appear in future. Nevertheless, SLOCCount is a proven tool and it has has been used on studies on Red Hat [14] and on Debian [4].

The results of the SLOCCount analysis are transformed afterward into other formats, including both relational and XML data formats. Hence, with a simple web interface anyone can have access to raw data and more elaborated visualization forms that facilitate a first analysis (graphs, maps, among others). Many of the results carried out for this study are offered in a web site[4].

4. RESULTS AND OBSERVATIONS

In the following subsections we are going to present and discuss the results obtained from applying our methodology to several Debian releases.

4.1 Observations on the size of Debian

At the time of publication, the latest stable release of Debian is version 3.1, also known under the codename *sarge*. The testing version has been codenamed *etch* and will become the next stable Debian version some time in the future. Finally, the one that is in development is called *sid*. In the past, *sarge* also passed through this testing phase. What we are going to consider in this work are the stable versions of Debian since version 2.0, published in 1998. Thus, we will consider Debian 2.0 (*hamm*), Debian 2.1 (*slink*), Debian 2.2 (*potato*), Debian 3.0 (*woody*) and, finally, Debian 3.1 (*sarge*). The codenames of the versions in Debian correspond to the main characters of the animated cartoon film *Toy Story*.

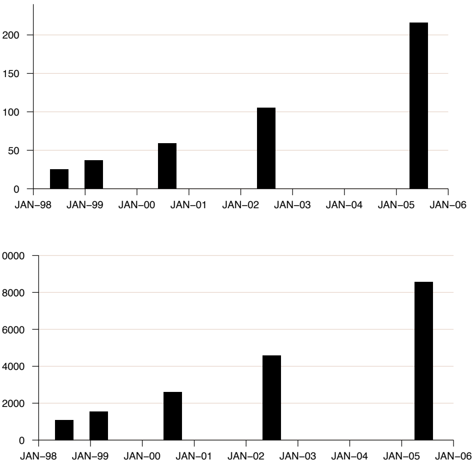

Figure 1: Size in MSLOC and number of packages for the versions in study. Top: MSLOC for each version. Bottom: Number of packages for each version. In both cases, the studied versions are spaced in time along the X axis according to their release date.

In figure 1 the number of MSLOC and source packages for the considered stable versions of Debian can be found. Debian 2.0, released July 1998, includes 1,096 source packages that have more than 25 MSLOC. The following stable version of Debian, version 2.1 (published around nine months later), contains more than 37 MSLOC in 1,551 source packages. Debian 2.2 (released 15 months after Debian 2.1) sums up around 59 MSLOC in 2,611 packages, whereas the next stable version, Debian 3.0 (published two years after Debian 2.2), groups 4,579 packages of source code with almost 105 MSLOC. Finally, almost three years later, Debian 3.1 has been released, with 8,560 source packages and more than 216 MSLOC.

Version	Release	Source pkgs	Size	Mean pkg size
2.0	Jul 1998	1,096	25	23,050
2.1	Mar 1999	1,551	37	23,910
2.2	Aug 2000	2,611	59	22,650
3.0	Jul 2002	4,579	105	22,860
3.1	Jun 2005	8,560	216	25,212

Table 1: Size of the Debian distributions under study. Size is given in MSLOC, while the mean package size is in SLOC.

[2] The original Sources.gz file where this entry comes from can be found at http://www.debian.org/mirror/list.

[3] http://www.dwheeler.com/sloccount/

[4] http://libresoft.dat.escet.urjc.es/debian-counting/

Although the number of points is not sufficient to make an accurate model, we can infer from the current data that the Debian distribution doubles its size (in terms of source lines of code and of number of packages) around every two years, although this growth has been much more significant at the beginnings (from July 1998 to August 2000 we observed an increase of 135%) than in later releases (between July 2002 and June 2005 the source code base has not achieved a 100% increase even though 3 years have passed). Hence, using time in the horizontal axis, we would have a smooth growth of the software compilation as found by Turski [13]. On the other hand, if we considered only releases (which is the methodology preferred by Lehman), the growth would be super-linear basically because the time interval between subsequent releases has been growing for most recent releases.

4.2 Observations on the size of packages

The histograms in figure 2 display package sizes for Debian 2.0 and Debian 3.0 (measured in SLOC). It can be clearly observed that large packages grow in size with time, while at the same time more packages near the origin appear. It is astonishing how many packages are *very small* packages (less than thousand lines of code), *small* (less than ten thousand lines) and *medium-sized* (between ten thousand and fifty thousand lines of code).

A small number of large packages in size (over 100 KSLOC) exist and the size of these packages tends to increase over time, as the sixth *law* of software evolution states [8]. Nevertheless, it seems surprising that in spite of the growth that Debian has undergone, the graph does not show big variations. Still more interesting is the fact that the mean size for the packages included in Debian is slightly regular (around 23,000 SLOC for Debian 2.0, 2.1, 2.2, 3.0 and 3.1, see table 1). With the data available at the present time it is difficult to give a solid explanation of this fact, but we can suggest some possible hypotheses[5]. As packages tend to grow in size and if no new packages are added to new versions of Debian, a growth in the mean package size would be expected. So it is the inclusion of new, small packages that makes the mean size stay almost constant for around seven years. Perhaps the *ecosystem* in Debian is so rich that while many packages grow in size, smaller ones are included causing that the average to stay approximately constant.

4.3 Observations on the maintenance of packages

Up to the moment, we have seen how Debian has been growing in the last 7 years as far as the number of packages and the number of SLOC is concerned. In the following paragraphs, we will attend an opposite dimension: packages that have not changed. This has to be understood in the sense that taking care of a software distribution requires maintaining packages, i.e. among other activities including new versions of the packages in the distribution. Packages that maintain from one release to the other the same version number may have been maintained actively, but usually even performing corrective maintenance implies releasing new versions of the software. We can therefore assume that no changes have been performed if the version number has not been changed.

It should be noted that Debian has a policy about version numbers. In addition to the software version, the project appends an own revision number. So, for example, a package with version number 1.2-3, means that it is the third Debian revision (upload) of that package in its 1.2 version. So, even if the original software is not maintained, the Debian versions may change because of library transitions (for instance, compiler changes from GCC 3 to GCC 4). Thus, some packages have Debian revision numbers up to 20 or higher - simply because the original software is not developed anymore but Debian still maintains that package.

Figure 4.3 will help explaining how we are going to measure maintenance activity supposing that we have two distributions (given each one by a set of packages, in the figure these are Debian 2.0 and Debian 3.1). The circle that gives the set of packages for the Debian 3.1 version has a larger radius as it contains many more packages than Debian 2.0 (the area of the circles could be considered as proportional to their size in number of packages). Both sets may have packages in common (the intersection between the two sets, as it is the case for the kernel-source package). Other packages will only be included in one of them. If packages appear only in the older Debian version, we say that it has been *lost*, while packages that appear only in the newer one are new (or added) packages. We can also identify a subset of those packages that remain with the same version number (a subset of the intersection between the two sets); those are the packages that we will consider unmaintained.

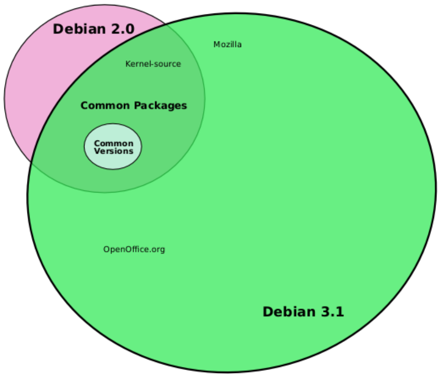

Figure 3: Illustration of common packages between Debian 2.0 and 3.1. Among these packages, we may find a subset that has the same version number.

Tables 2 and 3 contain some statistics about common packages in different stable versions. As explained above, we assume that two versions have a package in common if that package is included in both, independently of the version number of the package. Each table displays in its second column the number of packages that a version of Debian has in common with the other versions (see column "Com pkgs"). To facilitate the comparison in relative and absolute terms, the same version of Debian that is compared is included. Needless to say, Debian 2.0 will have in common with itself 1,096 (all) source packages.

Out of the 1096 packages included in Debian 2.0 only about 800 appear in the latest version of Debian (at time of

[5] One of the anonymous workshop reviewers pointed out that this kind of distribution is common in human-created artifacts, and is often considered to be an indication of human cognitive limitations.

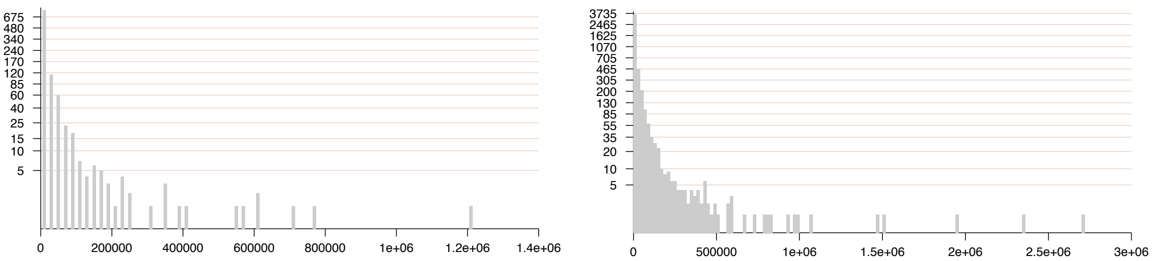

Figure 2: Histogram with the SLOC distribution for Debian packages. Left: Debian 2.0. Right: Debian 3.0

publication of this paper). This means that around 25% of the packages have disappeared from Debian in seven years. The number of packages of the 3.0 version that are still included in 3.1 is 3,848 out of 4,578 which gives us a similar percentage of *lost* packages.

If we consider those packages with version numbers that have not varied, we have to identify packages included in two different Debian versions that have the same package version (see column "Com vers."). Again, we add the own Debian version being compared. Because of that, Debian 2.0 will have all of its packages (1,096) in common with itself.

The fact that Debian 3.1 includes 158 packages that have not evolved since Debian 2.0 is very surprising, as 15% of the source packages included in Debian 2.0 have stayed almost with no alterations since they were introduced seven years ago (or earlier). As expected, the number of packages with versions in common increases for neighboring distributions.

4.4 Observations on the programming languages

Our methodology implies to identify the programming language of source code files before counting the number of SLOCs. Thanks to this, we are able to compute the significance of the different programming languages in Debian. The most used language in all Debian versions is C with percentages that vary between 55% and 85% and with a big advantage on its immediate pursuer, C++. It can be observed, nevertheless, that the importance of C is diminishing gradually, whereas other programming languages are growing at a steady rate.

For example, in table 4 the evolution of the most significant languages – those that surpass 1% of code in Debian 3.1 – is shown. Below the 1% mark we can find, in this order: tcl, Ada, PHP, Pascal, ML, Objective C, YACC, C#, Lex, Awk, Sed and Modula3.

There exist some programming languages that we could consider as minor languages and that reach a high position in the classification. This is because although being present in a reduced number of packages, these are large in size. That is the case of Ada, that sums up 430 KSLOC in three packages (gnat, an Ada compiler, libgtkada, a binding to the GTK library, and Asis, a system to manage sources in Ada) of a total of 576 KSLOC that have been identified as code written in Ada in Debian 3.0. A similar case is the one for Lisp, that counts with more than 1.2 MSLOC only for GNU Emacs and XEmacs of around 4 MSLOC in the whole distribution.

The programming language distribution pie-charts display a clear tendency in the decline in relative terms of C. Something similar seems to happen to Lisp, which was the third most used language in Debian 2.0 and has become the fifth in Debian 3.1 (in fact, in 3.1, the forth language is Perl), and that foreseeably will continue backing down in the future. In contrast, the part of the pie corresponding to C++, shell and other programming languages grows.

Figure 5 provides the relative evolution of programming languages which gives a new perspective of the growth for the last five stable Debian versions. We therefore take the Debian 2.0 version as reference and suppose that the presence of each language in it is 100% (normalized to 1) so that growth for a programming language is shown relative to itself. The graph should be read as follows: for each line in Debian 2.0 for a given language, the figure gives the number of lines in subsequent Debian releases for that language.

Previous pies evidenced that C is backing down as far as its relative importance is concerned. In this one we can observe that in absolute terms C has grown more than 300% throughout the four versions (see figure 4 for a histogram with absolute values). But we can see that scripting languages (shell, Python and Perl) have undergone an extraordinary growth, all of them multiplying their presence by factors superior to seven, accompanied by C++. Languages that grow a smaller quantity are the *traditional*, compiled ones (Fortran and Ada) and others (such as Lisp, a *traditional* language that does not require compilation). This can give an idea of the importance that interpreted languages have begun to have in the libre software world.

Figure 5 includes the most representative languages in Debian, but excludes Java and PHP, since the growth of these two has been enormous, in part because their presence in Debian 2.0 was testimonial, in part because their popularity in the latest time is beyond doubt.

4.5 Observations on the file sizes

It should be remarked that some of the most important programming languages have spectacular increases in their use, but that their mean file sizes remain generally constant (see table 5). Thus, for C the average length lies around 260 to 280 SLOC per file, whereas in C++ this value is located in an interval going from 140 to 185. We can find the exception to this rule in the shell language, that triples its mean size. This may be because the shell language is very singular: almost all the packages include something in shell for their installation, configuration or as *glue*. It is probable that this type of scripts get more complex and thus grow over the years.

Version	Com pkgs	Com vers.	SLOC com vers.	Files com vers.	SLOC com pkgs
Debian 2.0	1,096	1,096	25,267,766	110,587	25,267,766
Debian 2.1	1,066	666	11,518,285	11,5126	26,515,690
Debian 2.2	973	367	3,538,329	86,810	19,388,048
Debian 3.0	754	221	1,863,799	70,326	15,888,347
Debian 3.1	813	158	1,271,377	15,296	15,594,976

Table 2: Packages and versions in common for Debian 2.0

Version	Com pkgs	Com vers.	SLOC com vers.	Files com vers.	SLOC com pkgs
Debian 2.0	813	158	1,271,377	15,296	15,594,976
Debian 2.1	1,124	231	2,306,969	27,543	23,630,211
Debian 2.2	1,946	508	4,992,308	60,525	36,584,110
Debian 3.0	3,848	1,567	16,042,810	211,299	78,451,818
Debian 3.1	8,560	8,560	215,812,764	931,834	215,812,764

Table 3: Packages and versions in common for Debian 3.1

	2.0	% 2.0	2.1	% 2.1	2.2	% 2.2	3.0	% 3.0	3.1	% 3.1
C	19,371	76.7%	27,773	74.9%	40,878	69.1%	66.6	63.1%	120.5	55.8%
C++	1,557	6.2%	2,809	7.6%	5,978	10.1%	13.1	12.4%	36.4	15.8%
Shell	645	2.6%	1,151	3.1%	2,712	4.6%	8.6	8.2%	20.4	9.4%
Perl	425	1.7%	774	2.1%	1,395	2.4%	3.2	3.0%	6.4	2.9%
Lisp	1,425	5.6%	1,892	5.1%	3,197	5.4%	4.1	3.9%	6.8	3.1%
Python	122	0.5%	211	0.6%	349	0.6%	1.5	1.4%	4.1	1.9%
Java	22	0.1%	58	0.2%	183	0.3%	0.5	0.5%	3.8	1.7%
Fortran	494	2.0%	735	2.0%	1,182	2.0%	1,939	1.8%	2.7	1.3%

Table 4: Top programming languages in Debian. For Debian 2.0, 2.1 and 2.2 the sizes are given in KSLOC, for versions 3.0 and 3.1 in MSLOC.

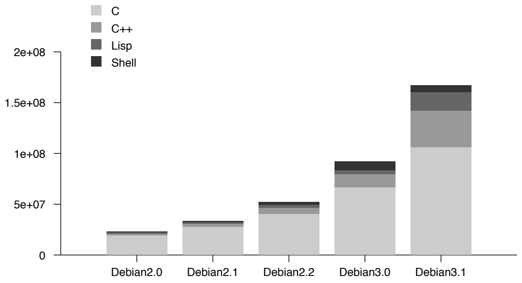

Figure 4: Evolution of the four most used languages in Debian.

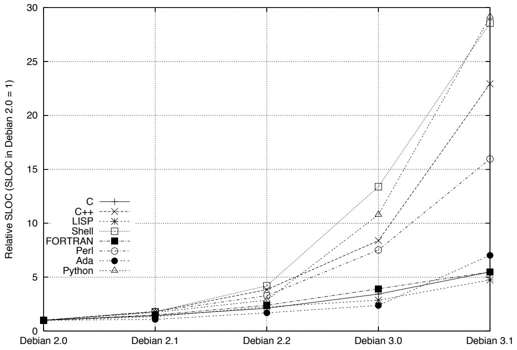

Figure 5: Relative growth of some programming languages in Debian.

It is very peculiar to see how structured languages usually have larger average file lengths than object-oriented languages. Thus the files in C (or Yacc) usually have higher sizes, in average, than those in C++. This makes us think that modularity of programming languages is reflected in the mean file size.

5. CONCLUSIONS AND FURTHER RESEARCH

In this paper we have shown the results of a study on the evolution of the stable versions of Debian from the year 1998 onwards. We have traced and presented the evolution of the size of its source code (measured in physical source lines of code), of the number and size of the packages, and of the use of the various programming languages.

Among the most important evidence we have found we can highlight the drastic evolution rate of the distribution: stable versions double in size (measured by number of packages or by lines of code) approximately every two years. This, when combined with the huge size of the system (about 200 MSLOC and 8,000 packages in 2005) may pose significant

Lang.	Deb. 2.0	Deb. 2.1	Deb. 2.2	Deb. 3.0	Deb. 3.1
C	262.88	268.42	268.64	283.33	276.36
C++	142.50	158.62	169.22	184.22	186.65
Lisp	394.82	393.99	394.19	383.60	349.56
shell	98.65	116.06	163.66	288.75	338.25
Yacc	789.43	743.79	762.24	619.30	599.23
Mean	228.49	229.92	229.46	243.35	231.6

Table 5: Mean file size for some programming languages.

problems for the management of the future evolution of the system, something that has probably influenced the delays in the release process of the last stable versions.

A specific problem in this realm comes from the fact that until now the mean size of packages has remained almost constant, which means that the system has more and more packages (growing linearly with the size of the system in SLOCs). Since there is a certain level of complexity related to the specifics of each package, which imposes a limit on the number of packages per developer, this means that the project would need to grow in terms of developers at the same pace. However, such a growth is not easy, and causes problems of its own, specially in the area of coordination.

With respect to the absolute figures, it can be noted that Debian 3.1 is probably one of the largest coordinated software collections in history, and almost for sure the largest one in the domain of general-purpose software for desktops and servers. This means that the human team maintaining it, which has also the peculiarity of being completely formed by volunteers, is exploring the limits of how to assemble and coordinate such a huge quantity of software. Therefore, the techniques and processes they employ to maintain a certain level of quality, a reasonable speed of updating, and a release process that delivers stable versions quite usable, are worth studying, and can for sure be of use in other domains which have to deal with large, complex collections of software.

As far as the programming languages are concerned, C is the most used language, although it is gradually losing importance. Scripting languages, C++ and Java are those that seem to have a higher growth in the newer releases, whereas the traditional compiled languages have even inferior growth rates than C. These variations also imply that the Debian team has to include developers with new skills in programming languages in order to maintain the evolving proportions. By looking at the trends in languages use withing the distribution, the project could estimate how many developers fluent in a given language it will need. In addition, this evolution of the different languages can also be considered as an estimation of how libre software is evolving in terms of languages used, although some of them are for sure misrepresented (for instance, Java is underrepresented, possibly because of licensing issues).

The evolution shown in this paper should also be put in the context of the activity of the volunteers doing all the packaging work. While some work has been done in this area [9], more research needs to be performed before a link can be established between the evolution of the skills and size of the developer population, the complexity and size of the distribution, the processes and activities performed by the project, and the quality of the resulting product. Only by understanding the relationships between all these parameters can reasonable measures be proposed to improve the quality of the software distribution, or shorten the release cycle without harming reliability and stability of the releases.

All in all, the study of distributions such as Debian can be of great interest not only for understanding their evolution, but also to be used as good case studies which can help to understand large, complex software systems which are more and more common in many domains.

6. REFERENCES

[1] J. J. Amor, J. M. Gonzalez-Barahona, G. Robles, and I. Herraiz. Measuring libre software using Debian 3.1 (Sarge) as a case study: preliminary results. Upgrade Magazine, Aug. 2005.

[2] H. Gall, M. Jazayeri, R. Klosch, and G. Trausmuth. Software evolution observations based on product release history. In *Proc Intl Conference on Software Maintenance*, pages 160-170, 1997.

[3] M. W. Godfrey and Q. Tu. Evolution in Open Source software: A case study. In *Proceedings of the International Conference on Software Maintenance*, pages 131-142, San Jose, California, 2000.

[4] J. M. Gonzalez-Barahona, M. A. Ortuno Perez, P. de las Heras, J. Centeno, and V. Matellan. Counting potatoes: the size of Debian 2.2. *Upgrade Magazine*, II(6):60-66, Dec. 2001.

[5] M. M. Lehman and L. A. Belady, editors. Program evolution: Processes of software change. *Academic Press Professional*, Inc., San Diego, CA, USA, 1985.

[6] M. M. Lehman and J. F. Ramil. Implications of laws of software evolution on continuing successful use of cots software. *Technical report*, Imperial College, 1998.

[7] M. M. Lehman and J. F. Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11(1):15-44, 2001.

[8] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution - the nineties view. In *METRICS'97: Proceedings of the 4th International Symposium on Software Metrics*, page 20, nov 1997.

[9] M. Michlmayr and B. M. Hill. Quality and the reliance on individuals in free software projects. In *Proceedings 3rd Workshop on Open Source Software Engineering*, pages 105-109, Portland, USA, 2003.

[10] G. Robles, J. J. Amor, J. M. Gonzalez-Barahona, and I. Herraiz. Evolution and growth in large libre software projects. In *Proceedings of the International Workshop on Principles in Software Evolution*, pages 165-174, Lisbon, Portugal, September 2005.

[11] G. Succi, J. W. Paulson, and A. Eberlein. Preliminary results from an empirical study on the growth of open source and commercial software products. In *EDSER-3 Workshop*, Toronto, Canada, May 2001.

[12] E. B. Swanson. The dimensions of maintenance. In *Proceedings of the 2nd International conference on Software Engineering*, pages 492-497, 1976.

[13] W. M. Turski. Reference model for smooth growth of software systems. *IEEE Transactions on Software Engineering*, 22(8):599-600, 1996.

[14] D. A. Wheeler. More than a gigabuck: Estimating GNU/Linux's size. *Technical report*, June 2001.