Empir Software Eng (2009) 14:262–285
DOI 10.1007/s10664-008-9100-x

Macro-level software evolution: a case study of a large software compilation

**Jesus M. Gonzalez-Barahona · Gregorio Robles ·
Martin Michlmayr · Juan José Amor ·
Daniel M. German**

Published online: 29 November 2008

Editors: Ahmed Hassan, Stephan Diehl and Harald Gall

Abstract Software evolution studies have traditionally focused on individual products. In this study we scale up the idea of software evolution by considering software compilations composed of a large quantity of independently developed products, engineered to work together. With the success of libre (free, open source) software, these compilations have become common in the form of 'software distributions', which group hundreds or thousands of software applications and libraries into an integrated system. We have performed an exploratory case study on one of them, Debian GNU/Linux, finding some significant results. First, Debian has been doubling in size every 2 years, totalling about 300 million lines of code as of 2007. Second, the mean size of packages has remained stable over time. Third, the number of dependencies between packages has been growing quickly. Finally, while C is still by far the most commonly used programming language for applications, use of the C++, Java, and Python languages have all significantly increased. The study helps not

J. M. Gonzalez-Barahona (✉) · G. Robles · J. J. Amor
Universidad Rey Juan Carlos, Madrid, Spain
e-mail: jgb@gsyc.es

G. Robles
e-mail: grex@gsyc.es

J. J. Amor
e-mail: jjamor@gsyc.es

M. Michlmayr
Open Source Program Office, HP, Innsbruck, Austria
e-mail: martin@michlmayr.org

D. M. German
University of Victoria, Victoria, Canada
e-mail: dmgerman@uvic.ca

only to understand the evolution of Debian, but also yields insights into the evolution of mature libre software systems in general.

Keywords Mining software repositories • Large software collections • Software evolution • Software integrators

1 Introduction

Software evolution studies usually consider single products developed by a coordinated team. However, software systems are commonly composed of a large set of applications and libraries, many of them coming from unrelated parties, and developed by different teams with their own goals. The evolution of those systems presents some specific aspects and characteristics that are worth studying. However, finding all the elements needed for such a study, and especially, the source code for the whole system at certain points of time, is not easy. This is probably the reason of the little attention paid to them by researchers in the area of software evolution.

Fortunately, the opportunity of performing such studies has become real with the advent of libre software[1] distributions: collections of software packages engineered to work in coordination, providing the user with a large operating system with many, maybe thousands, of applications. Each package is actually developed by a different group, usually called ‘project’, in relative isolation from the others. The job of creating and maintaining a distribution is mainly about making all packages fit together, and producing installers, package managers, some common look and feel, etc. Examples of libre software distributions are Fedora (Red Hat) Linux, FreeBSD, Ubuntu, and Debian GNU/Linux.

Although each package appears to be developed in isolation, there are relationships and interactions that become apparent when the whole system is considered. As a result, a dichotomy can be identified, similar to the one found in economics: software evolution in the small (the evolution of a single application) *versus* software evolution in the large (the evolution of compilations of software, composed of many different individual software applications that are combined together to form a system).

For this paper, we have selected one of the most popular libre software distributions, Debian GNU/Linux, and have examined it from a macro point of view. We have studied the stable releases of Debian over a period of nine years. For each release the source code of all applications was downloaded, and their evolution analyzed in terms of number of packages, size of each of them, programming languages used, and interdependencies among packages.

As a result of this analysis, we have found that Debian is an interesting collection composed of applications of varying sizes, with a large proportion of small, and few huge applications. Some of them evolve rapidly, while others change at a lower pace. Some applications have not changed during all the considered period, while others have been removed from the distribution. We have also discovered that, despite

[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).

being developed by different groups, applications are hardly isolated: they are subject to complex interdependencies that have to be satisfied for the whole system to work. The number of these dependencies tends to explode as the system grows, rendering it more difficult to maintain.

In Debian, developers must show confidence in the interest, usability and maturity of each package they select for the distribution. Given this selection criteria, a large share of all mature libre software ever available for Linux-based systems is present in it. Therefore, Debian can be considered as a good proxy of all mature libre software ever developed for such systems. This permits the interpretation of the results of this study in a larger framework, as an overview of the evolution of the landscape of libre software for Linux systems.

The rest of this paper is organized as follows. The next section introduces some of the main characteristics of libre software distributions, also showing previous research related to this study. Section 3 introduces and explains the main research questions addressed. Then, Section 4 details the methodology used for the collection and analysis of the data, with the intention of clarifying the results shown later, in Section 5, which is organized in six subsections: total size, size of packages, maintenance of packages, languages, file sizes, and dependencies. The paper ends with a section on conclusions and further research.

2 Libre Software Distributions and Related Research

Large distributions based on libre software are created in a manner that is quite different from traditional software development. In large non-libre software systems most of the work is performed in-house, with only some pieces licensed from other companies, and some work outsourced to third parties. Even in the case of intense outsourcing, the work is usually performed in close coordination under tightly defined requirements. Libre software, on the contrary, is typically written by small, independent teams of volunteers, sometimes collaborating with paid staff from one or more companies. While projects may interact with each other, in particular where dependencies between the software they produce exist, there is often no central coordination, neither common goals or guidelines.

Therefore, people building and maintaining software distributions are those who have to adapt each package to work in coordination with the rest. Usually, the required modifications are contributed back to the groups performing the actual development, in a continuous provision of feedback. Because of this, although usually they do not develop much code themselves, they have to know with some detail not only the general architecture of the software they are integrating, but also the development process used by the original team.

One of the most visible tasks performed by distributions is the automatic installation and management of packages. Manually installing and upgrading a libre software application is time-consuming and requires certain technical skills that not all users have, such as compiling or configuring the installation of the software. Doing that for the hundreds, if not thousands, of applications that are typically installed in a GNU/Linux system is out of question, even for experienced users, since it would require a significant effort to manually download and install (or upgrade, when a new release is available) each package. This is precisely the main role that distributions play: to select, test, and prepare applications so that they become packages easy to install, upgrade or remove. Unsurprisingly, a number of companies have found this to be a business opportunity, offering a distribution plus some related services, such as support. There are also various community distributions that operate on a non-profit basis like many other libre software projects.

In fact, public availability of the source code of libre software programs, and the possibility of freely redistributing them, has resulted in a large number of libre software distributions. Both characteristics also facilitate their study: several of them have been published, mainly for the well-known Red Hat and Debian systems. Those studies detail several parameters of the packages contained, their size, and some statistics on the programming languages present, among other issues (Wheeler 2001; Gonzalez-Barahona et al. 2001; Amor et al. 2005).

Among the different libre software distributions, Debian GNU/Linux has been selected for this study because it is one of the most popular, accessible, complete (in terms of number of packages maintained) and best established. Debian is a community effort that has provided a software distribution based on the Linux kernel for well over 12 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, all development infrastructure, including mailing lists, bug tracking, and the source code itself, is publicly available. In addition to integrating and maintaining software packages, members of the Debian project oversee the maintenance of a number of services, such as web sites and wikis.

Software evolution has been a matter of study for more than 30 years (Lehman and Belady 1985; Lehman and Ramil 2001). So far, the scope of software evolution analysis has always been single applications, such as the “classical” analysis of the OS/360 operating system (Lehman and Belady 1985) or, more recently, those on the Linux kernel (Godfrey and Tu 2000) or other well-known libre software applications, including Apache and GCC (Succi et al. 2001). Noteworthy is the proposal of studying the evolution of applications at the subsystem level (Gall et al. 1997), as this introduces the issue of granularity. Our approach considers a complete software compilation to be a system, with the constituent applications and libraries serving the role of subsystems.

The authors are not aware of a study on the evolution of a system integrating many independent software applications. In fact, software compilations have rarely been studied in software engineering, probably because of the constraints found when integrating software from different vendors, such as the restrictions imposed by the license of each piece. It is noticeable 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 paid to how the integration of these components evolve. A promising path in this direction has been the study of integration of COTS from a software evolution perspective (Lehman and Ramil 1998).

3 Research Questions

Software compilations are composed of heterogeneous pieces of software from many different sources. Some are developed by large, organized, well funded groups, such as the Mozilla Foundation, while some others are developed by a handful of volunteers. Therefore, they provide a diverse and comprehensive view of the libre software landscape. Furthermore, distributions provide a way to understand how different applications are interrelated (how each depends upon one or several other applications, and vice versa).

In this context, a compilation of the size of Debian can be considered a good proxy of libre software in general, thus offering a macroscopic view of the libre software landscape. Therefore, this paper can be considered to present a holistic study of libre software, analyzing how it is *in the large*, and drawing some conclusions about the phenomenon itself. Because of that, it is important to characterize the evolution of the main parameters of the distribution: total size (in lines of code), and total number of packages. The specific study of the distribution of the size provides some additional insight into this evolution.

Additionally, the changing demographics over time of programming language use presents itself as an avenue of exploration in this study. We examine the changes in popularity of various languages with respect to Debian applications, and discuss possible reasons for the various shifts and long term trends.

From another point of view, this paper goes a step beyond the *single-release* analysis of software distributions by considering their evolution over time. In this respect, the main goals slightly differ from those commonly found in software evolution studies. This is due to the different type of work involved in the creation and maintenance of software compilations, which is mainly integration and maintenance, with only some marginal true development. For example, a distribution might require an installer or some other software to perform administration tasks that require development effort, but theses cases are few in the context of all the effort to produce a new release. There are also some aspects that are common to classical software evolution analysis, such as how the size of the software it includes evolves.

Of course, software compilations must be maintained, but the practice of compilation maintenance differs from that of (single) software system maintenance. For example, Swanson's well known categories of corrective, adaptive, and perfective software maintenance (Swanson 1976) have little bearing on software compilations. Instead, software compilation maintenance focuses on the integration of new versions of software that have been released. In other words, package maintainers keep track of each software application, and update the distribution with the newer versions. They also check that the package keeps working when new releases of libraries and other programs used by it are updated. It is not uncommon for package maintainers to become bug-reporters of the applications they maintain. This raises interesting issues worth investigating, such as when packages are included or removed from the compilation, and the tracking of packages in several releases of the compilation, to learn about their evolution patterns.

From these observations, another important research question emerges: how much code is changing between Debian releases? This can be refined by studying both the size of the packages that remained unchanged between releases, and that of packages that were already present in previous releases, but have changed, at least in part.

The importance of the relationships between packages have already been stressed. Although they are developed and maintained by independent teams, with little or no coordination, in the end all of them have to work together. For performing its job, any application has a large set of requirements on a usually long list of packages that it uses in one form or another. Therefore, an important research question is also how those dependencies evolve, both from a macro (for the distribution as a whole) and a micro (for a specific application) point of view.

In the end, by understanding how all these aspects evolve, we address the general question of how the Debian software distribution is evolving. Since it is a good proxy of mature libre software available for Linux, some insight on its characterization is also obtained.

4 Methodology

Distributions are organized as a set of packages, each one usually corresponding to an application or a library, although they can also correspond to other products, such as documentation. As most libre software distributions do, Debian defines two different types of packages: source and binary. A source package contains the source code needed to produce a binary, installable, package. Once built, a source package results into one or more binary packages.

Debian maintains a *Sources* file for each release, describing the source packages that it contains. For each package, it contains the name and version, the list of binary packages built from it, the name and e-mail address of the maintainer, and some other information not relevant for this study. A package can be maintained by an individual or a team.

As an example, an excerpt of the entry for the *mozilla* source package in Debian 2.2 is included below,[2] showing that 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
[...]
```

For each Debian release, several binary distributions (collections of binary packages) are available, corresponding to the different architectures supported (for example: i386). Each binary distribution is defined by a list of descriptions of binary packages, found in the corresponding *Packages* file. The description of a binary package is similar to that of a source package, but contains some other fields of

[2] The original Sources file in which this entry can be found is in http://www.debian.org/mirror/list.

interest. For example, in Debian 2.2, the *mozilla* binary package is described as follows:

```
[...]
Package: mozilla
Priority: optional
Section: web
Installed-Size: 25428
Maintainer: Frank Belew (Myth) <frb@debian.org>
Architecture: i386
Version: M18-3
Replaces: mozilla-dmotif, mozilla-smotif
Provides: www-browser
Depends: libc6 (>= 2.1.2), libglib1.2 (>= 1.2.0),
    libgtk1.2 (>=1.2.7-1), libjpeg62, libpng2, libstdc++2.10,
    libz1, xlib6g (>= 3.3.6-4), libnspr4 (= M18-3), xcontrib
Recommends: mime-support
Suggests: postscript-viewer, pdf-viewer, eeyes |
    imagemagick | netpbm | xli | xloadimage | xv, xanim |
    ucbmpeg-play, freeamp | amp | splay | maplay | mpg123 | xmms
Conflicts: mozilla-dmotif, mozilla-smotif
Filename: dists/potato/main/binary-i386/web/mozilla_M18-3.deb
Size: 8941048
MD5sum: 739c13960dd8e62b7a677011cd0f86ab
Description: An Open Source WWW browser for X and GTK+
 Mozilla is a sophisticated graphical World-Wide-Web browser,
 with large
[...]
```

The *Depends* field of a package description lists other binary packages needed for it to run successfully. Therefore, packages that satisfy those dependencies should be installed before, or at the same time, than the described package. In the above case, each of the packages in the *Depends* field (in some cases specific versions of packages, such as a version of *libc6* higher or equal to 2.1.2, or *libnspr4* version M18-3) should be installed before, or at the same time, than *mozilla*. Each of these dependencies is either explicit (one and only one package is specified), *one-of-many* (a list of packages separated by | is specified, of which only one is required to be installed, for example `eeyes | imagemagick | netpbm |... |xv`), or an abstract dependency (an identifier for a common *one-of-many* dependency—e.g., *emacsen* is commonly used to indicate a choice of either version of *emacs* or *xemacs* to be installed). *Pre-Depends* is a similar field used by some packages, listing dependencies that should be installed before the installation of the package can proceed.

A Debian binary package may also have some optional requirements, listed in the *Recommends* and *Suggests* fields. The Debian Policy Manual defines packages listed in *Recommends* as strong but not required dependencies, and as those that would "be found together with this package in all but unusual installations." *Suggests* is used to declare optional dependencies that would enhance the original package but are not as common as those listed in the *Recommends* field. For a detailed formalization of Debian dependencies, and the method used to resolve them, see Mancinelli et al. (2006).

The study presented in this paper started by retrieving the files describing each Debian GNU/Linux stable releases between 2.0 and 4.0, that is: 2.0, 2.1, 2.2, 3.0, 3.1, and 4.0. For each of them, the corresponding *Sources* and *Packages* files of the i386 binary distribution were considered.

Once retrieved, *Sources* files were parsed, storing the resulting data into a database. Then, each source package was retrieved, the programming languages used in it identified, and the number of source lines of code (SLOC) for each file it contained, counted. The counting and language identification was performed with the SLOCCount tool. This tool analyzes a directory with source code, (in our case corresponding to a source package), identifies (by a series of heuristics) the files that contain source code, identifies for each of them (also by means of heuristics) the programming language, and finally counts the number of source lines of code they contain. SLOCCount counts "physical SLOC", defined as follows: "a physical source line of code (SLOC) is a line ending in a newline or end-of-file marker, and which contains at least one non-whitespace non-comment character.".[3]

SLOCCount also identifies identical files 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. MD5 detects identical files, but not those that have been slightly modified. With respect to automatic code, heuristics detect well-known or common cases, but may fail in some scenarios. Nevertheless, SLOCCount is a proven tool and it has been used on studies of Red Hat (Wheeler 2001) and Debian (Gonzalez-Barahona et al. 2001).

The results of the SLOCCount analysis were converted later into other formats, including both SQL and XML, which were used for later analysis, and for publishing most of the data.[4]

For creating the dependency graphs of each release, the corresponding *Packages* file were parsed, searching for *Depends*, *Pre-Depends*, *Suggests* and *Recommends* fields.

The relationship between packages and their dependencies can be modeled as a directed graph, where nodes are either binary packages or abstract dependencies; and edges correspond to their dependencies (see details in German 2007 and German et al. 2007). The edges are typed according to their importance: required (*Depends* and *Pre-Depends*) and optional (*Recommends* or *Suggests*). Abstract packages are connected to those that can "satisfy" them. A node can be further annotated with other attributes of the package, such as its license or installed size. We refer to this graph as the *Inter-Dependency Graph* (IDG) of the distribution. By extension, the IDG of a package p is the subset of the distribution's IDG that is reachable from p. Figure 1 shows the IDG of PostgreSQL under Debian 2.0.

For IDGs, the following notation has been used (German et al. 2007): the starting package is depicted as a circle; binary packages are depicted as rectangles; abstract dependencies are depicted as diamonds; and the packages that are always installed in a Debian system are colored in orange (darker). These graphs are similar to those defined by Mancinelli et al. (2006), being the main difference that they do not contain

[3] More details about this tool can be found in http://www.dwheeler.com/sloccount/.

[4] Available in http://libresoft.es/debian-counting/.

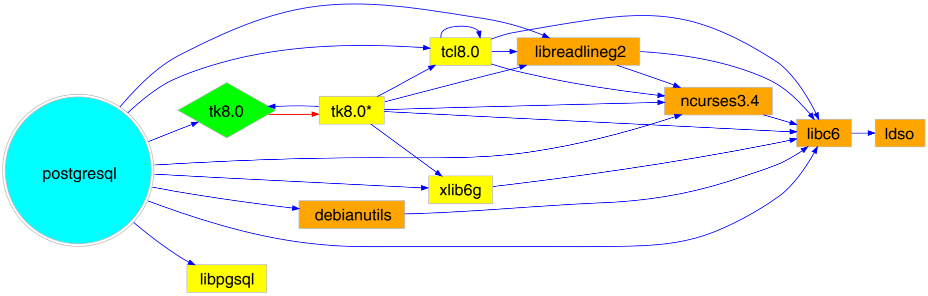


Fig. 1 *Inter-Dependency Graph* for *PostgreSQL* in Debian 2.0

nodes for abstract dependencies, which they call disjunctive dependencies. Instead, such information is stored as logic predicates. In addition, Mancinelli graphs include information about conflicting binary packages, and the nodes are annotated with the version of the package that they require.

Lets assume that the IDG of the distribution is $G = (V, E)$, where V is its set of nodes, and E its set of edges.

- *Direct dependencies* of package p is the set of nodes (binary packages and abstract dependencies) in V that are directly connected to p.
- *Direct subordinates* of package p correspond to those nodes in V (binary packages and abstract dependencies) from which there is an edge to p. The *Direct subordinates* of p have p as one of its *Direct dependencies*.

Abstract dependencies represent a choice of one of many packages; only one of them needs to be installed to satisfy the dependency. This implies that there might be multiple ways in which the dependencies of a package can be satisfied (Tucker et al. 2007). We define the *Instance* of the IDG of a package as a subset of its IDG where each abstract dependency points only to one package (the one that solved that abstract dependency). A specific instance of an IDG of a package p represents how p can be installed in a specific Debian system, with specific packages solving each abstract dependency.

The Debian Popularity Contest[5] surveys the usage of Debian packages, by tracking those actually installed by users participating in it. We use the Popularity Contest data to estimate the most likely way an abstract dependency is satisfied, computing the *popular instance* of the IDG (pIDG) of a package. This instance is computed selecting, for each abstract dependency that can be satisfied by several packages, the one with the highest popularity. Therefore, this instance includes no optional packages. Unfortunately the Debian Popularity Contest does not archive data for each Debian release, and we assume that the popularity of a package is the same across releases. When there is no popularity data for the options of an abstract dependency we choose the first listed, which is the algorithm used by Debian's

[5] http://popcon.debian.org/.

package system to decide how to resolve it if none of the options is already installed. We define two more sets:

- *All dependencies* of package p is the set of binary packages in its pIDG.
- *All potential subordinates* of package p is all binary packages in a Debian distribution that include p in its IDG.

The set of all dependencies of p corresponds to the most common set of applications that need to be installed before p can function (each abstract dependency—one-of-many, or disjunctive—is resolved to exactly one binary package). All potential subordinates, on the other hand, include any binary package that might require p. For example, in Debian 2.2 *mozilla* lists *xlib6g* as one of its direct dependencies; and *xlib6g* lists *xfree86common* as one of its direct dependencies. *xlib6g* and *xfree86common* are members of *all dependencies* of *mozilla*. At the same time, *mozilla* and *xlib6g* are members of the set of *all potential subordinates* of *xfree86common*. Figure 6 shows the pIDG of *mozilla* under Debian 2.2.

5 Results and Observations

In the following subsections, the main results obtained from the study presented in this paper are shown and discussed.

5.1 Total Size

The total size of the six studied releases of Debian is shown in Table 1. It presents, for each release, the date of publication, the total number of SLOC (sum of the SLOC of all packages in the distribution), the number of packages it contains, and the mean package size in SLOC. In nine years the number of packages in Debian and the total number of lines of code have grown by an order of magnitude, while the average size of a package has remained relatively stable.

Figure 2 shows the size of each distribution with respect to time. Although the number of points is insufficient to obtain a statistically significant model, we can infer from the current data that the Debian distribution has doubled in size in terms of source lines of code and of number of packages around every 2 years. This growth has been fastest at the beginning of the period: from July 1998 to August 2000 we observe an increase of 135%. In later releases this pace has slowed, and for example between July 2002 and June 2005 the source code base has not experimented a 100% increase during this 3 year period.

Table 1 Size, in number of source packages and total lines of code, and mean package size of the Debian releases studied

Release	Date	Source pkgs	Size (MSLOC)	Mean pkg size (SLOC)
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
4.0	Apr 2007	10,106	288	28,544

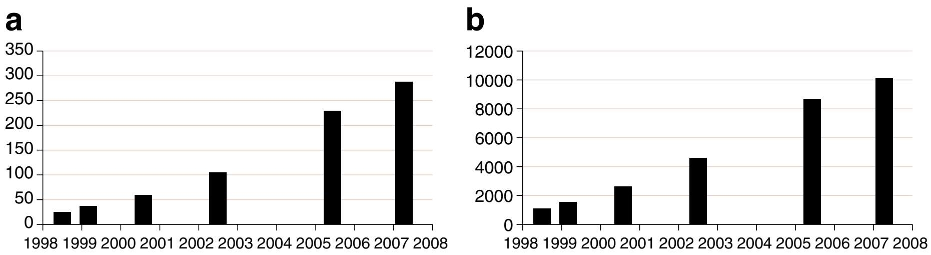


Fig. 2 Size in MSLOC (*left*) and number of packages (*right*) of the Debian releases studied. In both cases, releases are spaced in time along the X axis according to their publication date

In general terms, using time in the horizontal axis, a smooth growth of the software compilation can be observed, which is compatible with that described by Turski (1996). However, if we considered only releases, which is the methodology preferred by Lehman, the growth would be super-linear. The main reason for this is that the time interval between subsequent releases has been growing for most recent ones. However, given that the Debian project has not been actively seeking to increase the release interval, Turski's model seems more appropriate in this case.

5.2 Size of Packages

Histograms in Fig. 3 display package sizes for Debian 2.0 and Debian 3.0 (measured in SLOC). It can be observed that the largest packages are getting larger and larger, while at the same time more and more small packages enter the distribution. It is surprising how many packages are *very small* (less than one thousand lines of code), *small* (between one and ten thousand lines) and *medium-sized* (between ten and fifty thousand lines of code).

A small number of large packages (over 100 KSLOC) exist, with their size increasing over time, as the sixth *law* of software evolution predicts (Lehman et al. 1997). Perhaps the most significant fact is that the average size of packages is relatively stable, around 30 KSLOC for Debian 4.0 and 23 KSLOC for other releases, see Table 1. Currently, we lack an authoritative explanation for this phenomenon, but we have several hypotheses. One of them is that libre software production tends to grow mainly by creating new, more specialized, smaller packages (that can be

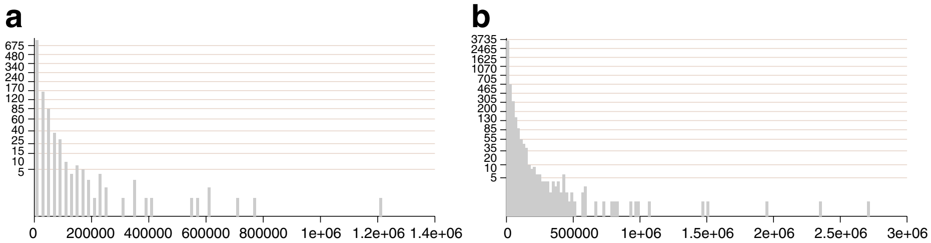


Fig. 3 Histograms of the distribution of the size (in SLOC) of packages in Debian releases 2.0 (*left*) and 3.0 (*right*)

Table 2 Number of packages (and SLOC of those packages) common and unchanged, for each release of Debian, with respect to release 2.0, and number of files in unchanged packages

Release	Packages		SLOC		Files unchanged
	Common	Unchanged	Common	Unchanged	
2.0	1,096	1,096	25,267,766	25,267,766	110,587
2.1	1,066	666	26,515,690	11,518,285	115,126
2.2	973	367	19,388,048	3,538,329	86,810
3.0	754	221	15,888,347	1,863,799	70,326
3.1	813	158	15,594,976	1,271,377	15,296
4.0	721	132	12,297,611	1,235,327	6,338

developed by a handful of developers), rather than large, complex ones (that require a large software development team). With time, some of the most successful small packages may attract more interest and developers, and start to grow. Perhaps the total mixture in Debian is so rich that while many packages grow in size, smaller ones are included causing the average to stay approximately constant.

5.3 Maintenance of Packages

Packages in Debian are identified by a name, a version (which should match the version of the package as defined by its original developers) and a Debian package revision number with the following format: ⟨package name⟩-⟨version number⟩-⟨revision number⟩. For example, in Debian 4.0 the package for Mozilla's Firefox is identified as `mozilla-firefox-2.0.0.3-1`, which corresponds to version 2.0.0.3, first revision of the Debian package (the revisions of the package are changes to the package specification, as described in Section 4). Except for dynamic libraries, Debian package names are rarely changed.[6] This allows us to track packages from one release of Debian to another.

One of the main tasks of Debian maintainers is to track new versions of software packages, re-package them, and update the package descriptions accordingly. Whenever a new version of a package is released (either a major release, or a minor one) it is updated, and its identifier changed. This allows the assumption that if the version of a package in Debian has not changed, then the original package has not changed enough to warrant a new package version.[7] It is also possible that the package is no longer maintained, but still useful to warrant its inclusion in a distribution.

For any given pair of Debian releases we can classify packages into three sets: *common* (those that appear in both distributions), *removed* (those that are in the older distribution, but not in the newer one), and *new* (those that appear in the newer distribution but not in the older one). *Common* packages include *unchanged* packages, those with the same version number in both distributions.

Tables 2 and 3 contain some statistics of *common* and *unchanged* packages in the different distributions. To facilitate the comparison in relative and absolute terms,

[6] The Debian policy requires the shared object name of a library to be part of the package name. This permits different versions of the library to coexist in the same computer.

[7] It is possible for a package to be in active development and yet not having released a new version in time to be included in a new Debian distribution. But this is unlikely, given that Debian distributions are released several years apart, while libre software projects tend to 'release-early, release-often'.

Table 3 Number of packages (and SLOC of those packages) common and unchanged, for each release of Debian, with respect to release 4.0, and number of files in unchanged packages

Release	Packages		SLOC		Files unchanged
	Common	Unchanged	Common	Unchanged	
2.0	721	132	12,297,611	1,235,327	6,338
2.1	995	188	16,999,000	1,759,679	8,772
2.2	1,710	388	28,597,414	3,855,356	18,781
3.0	3,236	1,020	63,947,430	10,173,837	42,474
3.1	7,300	3,843	171,406,036	59,235,197	258,537
4.0	10,106	10,106	288,461,743	288,461,743	1,198,735

the Debian release that is compared is also included. For instance, Debian 2.0 has in common with itself all its (1,096) source packages.

Out of the 1,096 packages included in Debian 2.0, 721 can be found in 4.0 (common packages). This means that only around 30% of the packages in Debian 2.0 were removed by the time Debian 4.0 was released, nine years later. For comparison, the number of packages of the 3.1 release that are still present in 4.0 is 7,300, out of a total of 10,106, which gives a similar percentage of removed packages.

With respect to unchanged packages, release 4.0 includes 132 with the same version number than they had in Debian 2.0. In other words, no less than 15% of the source packages included in Debian 2.0 are still the same in Debian 4.0, 9 years later.

Table 3 compares 4.0 with the previous releases. Even though a large percentage of Debian 2.0 remains unchanged in 4.0, such code is very small with respect to the current size of 4.0.

It is also important to notice that the number of files in unchanged packages, as presented in Tables 2 and 3, does no reflect the total number of files unchanged, which is higher: there are many files that do not change between Debian releases even when the version number of their package changes. Something similar can be said for the unchanged number of SLOC in those tables: it refers only to the size of packages that did not change. But outside those packages, many other files also did not change.

5.4 Programming Languages

Table 4 shows the evolution of the most significant languages, those that account for at least 1% of code in Debian 4.0 (C, C++, Shell, Java, Perl, LISP, Python, PHP). Below that 1% mark we find, in order of their relative shares, also for Debian 4.0: Fortran, Tcl, Ada, Ruby, ML, Objective C, YACC, C#, Haskell, Expect, Awk and Modula3. The aggregation of all the Assembler code found would make it the 8th language in Debian 4.0, but has been omitted from the table.

The most used language in all Debian releases is C, with a large difference over the second, C++. However, the evolution of their shares for the first and last releases analyzed, falling from 77% to 51% in the case of C, raising from 6% to 19% for C++, show different stories. While the relative importance of C is diminishing gradually, that of C++, and other languages, is growing from release to release. It can also be noticed that despite of these trends, the absolute size of the code written in C has been growing for all releases from about 19 MSLOC in Debian 2.0 to more than

Table 4 Top programming languages in Debian 4.0, in MSLOC, for each Debian release studied, sorted by their importance in Debian 4.0

	2.0	%	2.1	%	2.2	%	3.0	%	3.1	%	4.0	%
C	19.37	76.7%	27.77	74.9%	40.87	69.1%	66.6	63.1%	120.5	55.8%	145.2	51.3%
C++	1.55	6.2%	2.80	7.6%	5.97	10.1%	13.1	12.4%	36.4	15.8%	52.9	18.7%
Shell	0.64	2.6%	1.15	3.1%	2.71	4.6%	8.6	8.2%	20.4	9.4%	29.3	10.3%
Java	0.02	0.1%	0.05	0.2%	0.18	0.3%	0.5	0.5%	3.8	1.7%	9.0	3.1%
Perl	0.42	1.7%	0.77	2.1%	1.39	2.4%	3.2	3.0%	6.4	2.9%	8.0	2.8%
LISP	1.42	5.6%	1.89	5.1%	3.19	5.4%	4.1	3.9%	6.8	3.1%	7.7	2.7%
Python	0.12	0.5%	0.21	0.6%	0.34	0.6%	1.5	1.4%	4.1	1.9%	7.2	2.5%
PHP	0.01	0.0%	0.01	0.0%	0.03	0.0%	0.6	0.6%	2.1	0.9%	3.3	1.1%

147 MSLOC in Debian 4.0. It just happens that it is not growing as quickly as other languages.

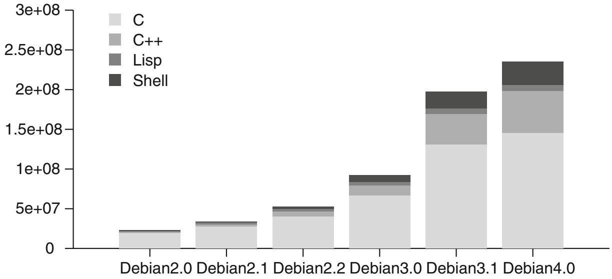
The case of Shell, in a solid third place, has mainly to do with its presence in almost any package, of any kind. With the entry of increasingly smaller packages in the latest Debian releases, all with some Shell code, the total share of Shell is growing accordingly.

The most rapid entry in this top-8 of the languages in Debian is certainly Java, which grows from a marginal 0.5% in Debian 3.0 to 1.7% in 3.1 and 3.1% in Debian 4.0. Although it is still far from the top-three languages, it is currently in a strong fourth position. The main reason for this is the availability of large applications, such as Eclipse or Azureus. It is important to notice that the releases under study do not include neither the Sun Java Runtime Environment nor the Sun Java Development Kit, due to licensing issues. Although there are other Java runtimes and development kits, it is quite possible that this causes an underrepresentation of Java, since for most other languages, Debian includes at least one of the usual development (compiling or scripting) environments.

To better understand the evolution of some of the top languages, (Fig. 4) shows the lines of code for four of them, for each studied Debian release. In it, the decline in relative terms of C, but its growth in absolute terms, is clearly visible. A noteworthy similar case is LISP, which is the third most used language in Debian 2.0 and becomes fourth in Debian 3.1 and fifth in Debian 4.0. In contrast to these, both Shell and C++ grow significantly, amounting for a large share of all the code in Debian 4.0.

Figure 5 provides a view of the relative evolution of some programming languages, normalizing to their respective situation in Debian 2.0. The relative SLOC (vertical axis) is computed by dividing the number of SLOC in a given distribution by the

Fig. 4 Number of SLOC in each Debian release for four of the top languages



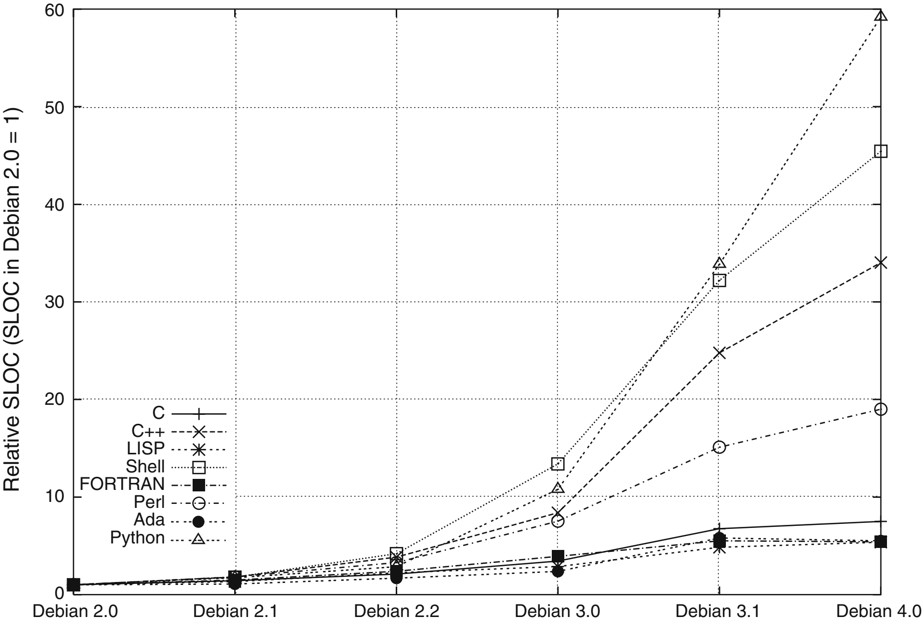


Fig. 5 Relative growth of some programming languages

number of SLOC in Debian 2.0. For example, Python has 60 times more SLOC in 4.0 than in 2.0, but C only seven times more. This plot is useful to identify some of the languages that have become more popular in the last nine years: Python, Shell, and C++ (Java is not in the figure). When this information is combined with Table 4, it is found that the growth of these languages is mainly at the expense of C and Perl.

Yet in absolute terms C has grown three times during this period, although the total number of SLOC has grown 10 times. At the same time, some scripting languages (Shell, Python and Perl) have undergone an extraordinary growth, all of them multiplying their presence by factors superior to seven.

In terms of SLOC, some programming languages that could be considered as uncommon account for a significant share of the distribution. This is because, even though they are present in a small number of packages, these packages are large. For example, Ada accounts for a total of 576 KSLOC in Debian 3.0. But 430 KSLOC come from three packages (Gnat, an Ada compiler; libgtkada, a language binding for the GTK+ library; and Asis, a system to manage sources of Ada programs). LISP follows a similar pattern: it accounts for approximately 4 MSLOC in Debian 3.0, but 1.2 of these come from two single packages: GNU Emacs and XEmacs.

5.5 File Sizes

The mean file size for most of the languages, including those with a largest share, show a remarkable stability from release to release (see Table 5).

This is especially noteworthy taking into account the large differences in SLOC for those languages in each release. For example, for C the mean length lies between 260 and 295 SLOC per file, whereas in C++ this value is between 140 and 196. An

Table 5 Mean file size for some programming languages

Lang.	2.0	2.1	2.2	3.0	3.1	4.0
C	262.88	268.42	268.64	283.33	276.36	295.29
C++	142.50	158.62	169.22	184.22	186.65	195.82
LISP	394.82	393.99	394.19	383.60	349.56	346.33
Shell	98.65	116.06	163.66	288.75	338.25	389.99
YACC	789.43	743.79	762.24	619.30	599.23	615.55
Mean	228.49	229.92	229.46	243.35	231.6	240.72

exception to this behavior can be observed for the Shell language, which has tripled its size from Debian 2.0 to Debian 4.0. This may be because the Shell language is peculiar: almost all packages include something in Shell for their installation, configuration or as *glue code*. It is likely that what happens is that these scripts get more complex over time, and thus grow over the years. This adds up to the fact that Shell programs are seldom divided into several files: if there is more functionality, usually they just get longer.

It is also remarkable how procedural languages usually have larger average file lengths than object-oriented languages. For example, the files in C or YACC are usually larger, in average, than those in C++. This suggests that class-inheritance or other characteristics of object-oriented languages are somehow reflected in shorter file sizes.

Table 6 Number of dependencies and subordinates of binary applications in different Debian releases

Release	2.0	2.1	2.2	3.0	3.1	4.0
Binary packages	1524	2269	3889	8273	15196	18042
All dependencies						
Median	3	4	5	6	17	21
Mean	4.32	5.26	8.33	12.33	30.09	35.83
Std. Dev.	3.37	4.70	10.01	15.92	35.01	42.31
Max	19	46	124	170	479	561
Direct dependencies						
Median	2	2	2	2	2	3
Mean	1.98	2.30	2.71	3.32	4.12	4.72
Std. Dev.	1.63	2.31	3.14	4.26	5.35	6.77
Max	15	20	47	70	106	86
All subordinates						
Median	0	0	0	0	0	0
Mean	5.23	6.15	10.58	17.25	51.50	75.67
Std. Dev.	52.01	66.21	98.65	152.56	465.31	732.51
Max	1357	2010	3443	7432	13702	16247
Direct subordinates						
Median	0	0	0	0	0	0
Mean	2.08	2.38	2.61	3.35	3.75	4.71
Std. Dev.	26.61	32.52	40.61	58.17	70.57	82.60
Max	971	1422	2329	4606	7605	8882

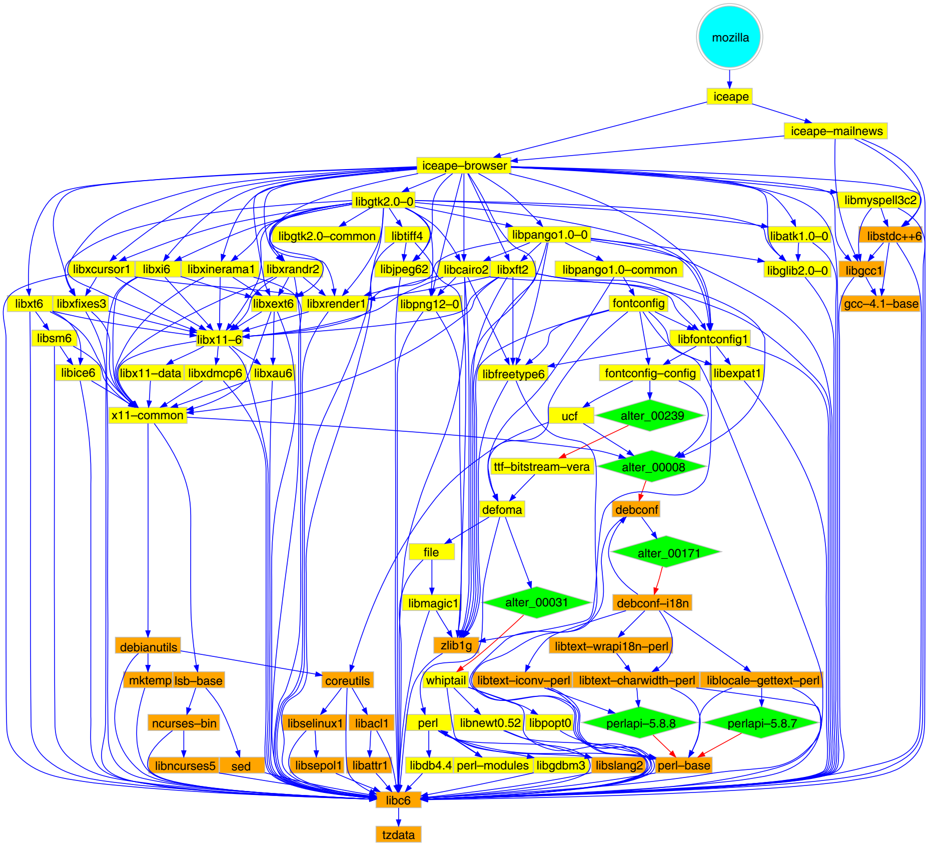


Fig. 6 Most popular instance of the *Inter-Dependency Graph* for *mozilla* in Debian 4.0. *mozilla* is the leftmost circle

5.6 Dependencies

Libre software, just like any other type of software, is designed to be modular. Software reuse is particularly easy in libre software, as there are no economic constraints: most libre projects can use the results of other libre projects without having to pay for that privilege. The only requirement is for the license of the module to be used to be consistent with the license that wants to use it. For example, GPL-licensed software is able to use a BSD-licensed library without any extra arrangements. See Rosen (2004) for a discussion of the main libre source licenses and their compatibility.

As was explained in Section 4, these relationships can be found, in the form of dependencies, in the Debian distribution. Table 6 summarizes the sizes of the dependents, one level dependents, dependencies, and one-level dependencies for the packages in the different releases of Debian. The number of binary packages in Debian has grown an order of magnitude from version 2.0 to 4.0. At the same time the mean number of all dependencies has grown at a similar rate: binary packages are becoming more interrelated.

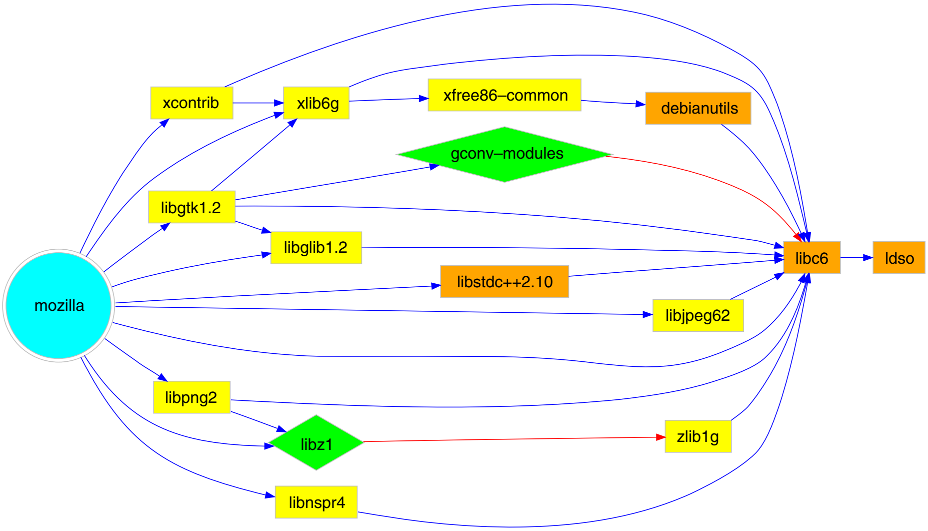


Fig. 7 Most popular instance of the *Inter-Dependency Graph* for *mozilla* in Debian 2.2. Each of the two abstract dependencies have only one child

In Debian 2.0 the packages with more dependencies had 19 (*python-gdk-imlib*, *boot-floppies* and *libhdf4-altdev*). In Debian 4.0 the package with the largest number of dependencies is *kde*, with 561, followed by *gnome*, with 486. *kde* and *gnome* are sets of GUI applications for the Unix desktop, none of them is present in Debian 2.0.

Both *kde* and *gnome* are bundles of packages. In practical terms this means that they do not have any source code associated: when these packages are installed, the bundle is installed. This raises three noteworthy issues: first, from the point of view of the user installing such bundles, these collections of packages operate as a single software product; second, it can be argued that these packages inflate the average number of dependencies without adding any new source code themselves; and third, they can be considered a great demonstration of the power of component-oriented software engineering, where a "new" application, the bundle, can be created from many components without writing a single line of code.

As the number of dependencies of packages evolves, their dependency graphs are likely to change too. For example, Fig. 6 shows the pIDG of *mozilla* in Debian 4.0, which can be compared to its dependency graph in Debian 2.2, depicted in Fig. 7. Mozilla required 13 packages in 2.2 (the first version of Debian to include it), and 72 in 4.0. This growth is expected as applications evolve and grow to satisfy newer requirements.

Table 7 Evolution of the number of all dependencies for some selected binary packages, for the studied Debian releases

	2.0	2.1	2.2	3.0	3.1	4.0
Apache1.3	6	7	11	45	57	64
Mozilla	N/A	7	13	21	75	72
PostgreSQL	9	16	9	23	54	42

Table 8 Evolution of the number of all potential subordinates for selected binary packages

Release	2.0	2.1	2.2	3.0	3.1	4.0
libc6	971	1422	2329	4606	7605	8882
perl	78	139	6	662	1419	1662
python[a]	19	46	65	160	452	777

[a] *python* was known as *python-base* from 2.0 to 2.2

With respect to the number of subordinates of a package, the story is different. In this case, the median is zero, meaning that most packages do not have any subordinates. Yet their average number keeps growing at a rate similar to the growth of the number of packages in the distribution. This implies that the subordinates of some packages are growing very fast (a small portion of packages are being used by a very large number of packages). For example, in Debian 2.0, *perl* has a total of 118 subordinates, but in Debian 4.0 it has 11,459. It is also not surprising that the packages with the largest number of subordinates are libraries (such as *libc6*, the GNU C library, which has the largest number of subordinates in every release of Debian), interpreters (such as *perl*) or utilities (such as *binutils* and *sed*). The number of potential subordinates can serve as a good indicator of the success of a library: the more binary packages that depend on it, the more successful it is. Table 7 and 8 show the evolution of the size of the dependencies and subordinates of selected applications.

In Subsection 5.2 it was highlighted how many of the newer applications are very small. It is now possible to argue that applications can be smaller because there are more packages, including libraries, available, upon which they can depend and reuse. In other words, applications can be smaller, but at the same time they can be more powerful.

6 Conclusions and Further Research

In this paper we have shown the results of a study on the evolution of the stable releases of Debian from 1998 to 2007. We have analyzed and presented the evolution of the size of their source code (measured in lines of source code), the number and size of their packages, the changed and unchanged packages, the use of programming languages, and the dependencies between packages.

Of the many findings from this study, one observation in particular stands out: stable releases 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 300 MSLOC and 10,000 packages in 2007) may pose significant problems for the management of its future evolution, something that has probably influenced the delays experienced for the last stable releases.

During the period under study, 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 SLOC. Debian 4.0 has 10 times more packages than Debian 2.0. In order to cope with this growth, Debian must increase its number of package maintainers, the number of packages under the responsibility of each maintainer, or both. Such a growth, however, is not easy to cope with, and causes problems of its own, especially in the area of coordination.

With respect to the absolute figures, it can be noted that Debian 4.0 is probably one of the largest coordinated software collections in history, and almost certainly 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 usable stable versions, 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 programming languages are concerned, C is the most commonly used, although it is gradually losing its dominance. Scripting languages (Perl, Python, Shell), C++ and Java are those with a higher growth in the newer releases, whereas most other compiled languages have even inferior growth rates than C. These variations also imply that the Debian team has to include developers with skills in new (for Debian) programming languages in order to maintain the evolving shares. Although Debian maintainers do not develop the packages themselves, they must have a detailed understanding of the their internal workings. Consequently, proficiency in the native programming language is a de facto necessity for them. By looking at the trends in languages used within the distribution, the project could estimate how many developers fluent in a given language will be needed. In addition, the evolution of the different languages can also be considered as an estimate of how libre software is evolving in terms of languages used, although some of them, such as Java, are certainly misrepresented.

One of the most surprising results has been the high number of packages that are present in the latest release exactly as they were in Debian 2.0, 9 years before. In general, the presence of unchanged packages between any two releases has been studied in detail, finding that there is a large share of them (with respect to the common packages, that is, those present with the same name in both releases). This indicates that a large share of the code in Debian did not require to be maintained for long periods of time, or maintenance was not performed on it, but the package still was found to have sufficient quality to be included in the distribution.

Using dependency information, we have shown that packages are highly interrelated, and as Debian evolves, the total number of dependencies grows quickly. We have also seen how packages with the interpreters for some scripting languages, Perl and Python, are among those being used by more packages, and that the C run time library, *libc6*, is being required by almost every package.

From a combination of the dependency information and the study of the size of the packages, we have learned that the growing number of small packages is possible because they can use many other components in the distribution. That is, the modularity of Debian, understood as a large collection of components, is allowing developers to build powerful, yet small applications, that gain advantage of using tens of other packages.

In summary, the study of large libre software distributions such as Debian has proven to be revealing, not only of how they are evolving over time, but also of how individual applications interact among themselves. The latter finding shows how these distributions, where applications and libraries are really ready to be used by any other application, foster the composition and code reuse at a new level. This kind of result emerges only after studying the system as a whole, although it mainly impacts how individual applications are built.

As further work, several research lines have been opened by this study. For example, the evolution of code artifacts shown in it could 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 (Michlmayr and Hill 2003), 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, reasonable measures can be proposed to improve the quality of the software distribution, or to shorten the release cycle without harming reliability and stability of the releases.

Another promising line is related to the further study of the evolution of dependencies. The trustability of an application depends not only on its own characteristics, but also on those of all the components (packages, in our case) that it is using. Therefore, there should be a balance between the convenience of using more and more external packages, for functionality, modularity and code reuse reasons, and the convenience of not using too much. Or at least, consider carefully how they impact on the trustability of the whole application. This balance could be studied over time, relating the different packages in the dependency set to bug reports and their relevance.

In general, all studies that relate the kind of analysis shown in this paper to other sources of information, such as the issue tracking databases of the projects, the mailing lists used for maintenance of the packages, the usage information available from the Debian Popularity Contest, etc., will allow for more interesting results. In the end, if Debian and other distributions are to be conceived as a rich ecosystem, more research is needed before we can model the relationship between their more important parameters.

Acknowledgements We thank the anonymous reviewers for their helpful comments and suggestions.

References

Amor JJ, Gonzalez-Barahona JM, Robles G, Herraiz I (2005) Measuring libre software using Debian 3.1 (Sarge) as a case study: preliminary results. Upgrade Magazine, Aug

Gall H, Jazayeri M, Klösch R, Trausmuth G (1997) Software evolution observations based on product release history. In: Proc intl conference on software maintenance, Bari, October 1997, pp 160–170

German DM (2007) Using software distributions to understand the relationship among free and open source software projects. In: 4th international workshop on mining software repositories (MSR 2006), Shanghai, May 2007

German DM, Gonzalez-Barahona JM, Robles G (2007) A model to understand the building and running inter-dependencies of software. In: Proc. 14th working conference on reverse engineering, Vancouver, 29–31 October 2007, pp 130–139

Godfrey MW, Tu Q (2000) Evolution in Open Source software: a case study. In: Proceedings of the international conference on software maintenance, San Jose, 11–14 October 2000, pp 131–142

Gonzalez-Barahona JM, Ortuno Perez MA, de las Heras P, Centeno J, Matellan V (2001) Counting potatoes: the size of Debian 2.2. Upgrade Mag II(6):60–66, Dec

Lehman MM, Belady LA (eds) (1985) Program evolution: processes of software change. Academic, San Diego

Lehman MM, Ramil JF (1998) Implications of laws of software evolution on continuing successful use of cots software. Technical report, Imperial College, June. http://www.doc.ic.ac.uk/research/technicalreports/1998/DTR98-8.pdf

Lehman MM, Ramil JF (2001) Rules and tools for software evolution planning and management. Ann Softw Eng 11(1):15–44

Lehman MM, Ramil JF, Wernick PD, Perry DE, Turski WM (1997) Metrics and laws of software evolution—the nineties view. In: METRICS '97: proceedings of the 4th international symposium on software metrics, Albuquerque, November 1997, p 20

Mancinelli F, Boender J, di Cosmo R, Vouillon J, Durak B, Leroy X, Treinen R (2006) Managing the complexity of large free and open source package-based software distributions. In: ASE '06: Proceedings of the 21st IEEE/ACM international conference on automated software engineering. IEEE Computer Society, Washington, DC, pp 199–208

Michlmayr M, Hill BM (2003) Quality and the reliance on individuals in free software projects. In: Proceedings of the 3rd workshop on open source software engineering, Portland, May 2003, pp 105–109

Rosen L (2004) Open source licensing: software freedom and intellectual property law. Prentice Hall, Englewood Cliffs

Succi G, Paulson JW, Eberlein A (2001) Preliminary results from an empirical study on the growth of open source and commercial software products. In: EDSER-3 Workshop, Toronto, May 2001

Swanson EB (1976) The dimensions of maintenance. In: Proceedings of the 2nd international conference on software engineering, San Francisco, 13–15 October 1976, pp 492–497

Tucker C, Shuffelton D, Jhala R, Lerner S (2007) OPIUM: optimal package install/uninstall manager. In: ICSE '07: Proceedings of the 29th international conference on software engineering. IEEE Computer Society, Washington, DC, pp 178–188

Turski WM (1996) Reference model for smooth growth of software systems. IEEE Trans Softw Eng 22(8):599–600

Wheeler DA (2001) More than a gigabuck: estimating GNU/Linux's size. June. http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html

Jesus M. Gonzalez-Barahona teaches and researches in Universidad Rey Juan Carlos, Mostoles (Spain). His research interests include libre software development, with a focus on quantitative and empirical studies, and distributed tools for collaboration in libre software projects. He works in the GSyC/LibreSoft research team, http://libresoft.es.

Gregorio Robles is Associate Professor at the Universidad Rey Juan Carlos, where he earned his PhD in 2006. His research interests lie in the empirical study of libre software, ranging from technical issues to those related to the human resources of the projects.

Martin Michlmayr has been involved in various free and open source software projects for well over 10 years. He acted as the leader of the Debian project for two years and currently serves on the board of the Open Source Initiative (OSI). Martin works for HP as an Open Source Community Expert and acts as the community manager of FOSSBazaar. Martin holds Master degrees in Philosophy, Psychology and Software Engineering, and earned a PhD from the University of Cambridge.

Juan José Amor has a M.Sc. in Computer Science from the Universidad Politécnica de Madrid and he is currently pursuing a Ph.D. at the Universidad Rey Juan Carlos, where he is also a project manager. His research interests are related to libre software engineering, mainly effort and schedule estimates in libre software projects. Since 1995 he has collaborated in several libre software organizations; he is also co-founder of LuCAS, the best known libre software documentation portal in Spanish, and Hispalinux, the biggest spanish Linux user group. He also collaborates with Barrapunto.com and Linux+.

Daniel M. German is associate professor of computer science at the University of Victoria, Canada. His main areas of interest are software evolution, open source software engineering and intellectual property.