Debsources: Live and Historical Views on Macro-Level Software Evolution*

Matthieu Caneill
Polytech Grenoble
Université Joseph Fourier, France
matthieu.caneill@e.ujf-grenoble.fr

Stefano Zacchirolı
Univ Paris Diderot, Sorbonne Paris Cité
PPS, UMR 7126, CNRS, F-75205 Paris, France
zack@pps.univ-paris-diderot.fr

ABSTRACT

Context. Software evolution has been an active field of research in recent years, but studies on macro-level software evolution—in particular, on the evolution of large software collections over many years—are scarce, despite the increasing popularity of intermediate vendors as a way to deliver software to final users.

Goal. We want to ease the study of both day-by-day and long-term Free and Open Source Software (FOSS) evolution trends at the macro-level, focusing on the Debian distribution as a proxy of relevant FOSS projects.

Method. We have built Debsources, a software platform to gather, search, and publish on the Web all the source code of Debian and measures about it. We have set up a public Debsources instance at http://sources.debian.net, integrated it into the Debian infrastructure to receive live updates of new package releases, and written plugins to compute popular source code metrics. We have injected all current and historical Debian releases into it.

Results. The obtained dataset and Web portal provide both long-term views over the past 20 years of FOSS evolution and live insights on what is happening at sub-day granularity. By writing simple plugins (∼100 lines of Python each) and adding them to our Debsources instance, we have been able to easily replicate and extend past empirical analyses on metrics as diverse as lines of code, number of packages, and rate of change—and make them perennial. We have obtained slightly different results than our reference study, but confirmed the general trends and updated them in light of 7 extra years of evolution history.

Conclusions. Debsources is a flexible platform to monitor large FOSS collections over long periods of time. Its main instance and dataset are valuable resources for scholars interested in macro-level software evolution.

*This work has been partially performed at, and supported by IRILL http://www.irill.org. Unless noted otherwise, all URLs and data in the text have been retrieved on March 9, 2014.

Categories and Subject Descriptors
D.2.8 [Software Engineering]: Metrics—product metrics;
H.4 [Information Systems Applications]: Miscellaneous;
K.2 [History of Computing]: [Software]

General Terms
measurement

Keywords
software evolution, source code, free software, open source, Debian

1. INTRODUCTION

For several decades now [21, 18] software evolution has been an active field of research. Given its natural availability and openness, numerous empirical studies on software evolution have targeted Free and Open Source Software (FOSS), with more than 100 noteworthy papers cited in recent systematic literature reviews [27, 3]. Despite the abundant research efforts, few studies have investigated macro-level software evolution (or “evolution in the large”), i.e., have considered large software collections as coherent wholes and observed their evolution, as collections, rather than the evolution of individual software products contained therein.

This lack of studies is not due to a lack of interest in studying software collections. To begin with, their relevance w.r.t. current practices is hard to dispute: with the massive popularization of “app stores” and the steady market share of package-based software distributions, software is increasingly delivered to users as part of curated collections maintained by intermediate software vendors. Additionally, software collections are also useful to study evolution at the granularity of individual software products: they contribute to eliminate (researcher) selection bias, which is often cited as the main threat to validity in evolution studies [27]. Finally, well-established software collections are enjoying remarkably long lives—now spanning several decades—outliving many of the software products they ship; software collections therefore offer remarkable opportunities for gathering long-term historical insights on the practice of software.

The study of software collections, however, poses specific challenges for scholars, due to an apparent tendency at growing ad hoc software ecosystems, made of homegrown tools, technical conventions, and social norms that might be hard to take into account when conducting empirical studies. We believe that the relative scarcity of macro-level evolution...
Contributions. We focus on Debian,\footnote{http://www.debian.org} one of the most reputed and oldest (founded in 1993) FOSS distributions, often credited as the largest organized collection of FOSS, and a popular data source for empirical software engineering studies (e.g., \cite{28,11,19,9}). Our aim is to ease the study of macro-level FOSS evolution patterns, using the assumption that Debian is a representative sample of relevant FOSS projects. More specifically, we want to support both long-term evolution studies—looking back as far as possible—as well as studies of present, day-by-day evolution patterns of software currently shipped by Debian.

To that end we have built Debsources, a software platform to gather, search, and publish on the Web the source code of Debian and measures about it. We have set up a Debsources instance at \url{http://sources.debian.net}, integrated it into the Debian infrastructure to receive live updates of new packages, and injected all current and historical Debian releases into it. To assess the usefulness of the platform we have used the obtained dataset to replicate the major studies on macro-level software evolution \cite{24,11} which, as it happens, targeted Debian too.

Debsources has made the data gathering process very easy. Thanks to its extensible design we just had to write a few short Python plugins to compute classical software metrics, trigger an update, and wait a few days to obtain the dataset. As a consequence of us doing so, the dataset needed to replicate the original studies is now live and perennial. Each Debian package release gets immediately processed by our plugins and the obtained results augment the dataset publicly available at our Debsources instance, which has quickly gained popularity in the Debian community.

Debsources is Free Software\footnote{http://anonscm.debian.org/gitweb/?p=qa/debsources.git} released under the AGPL3 license. It can be deployed elsewhere to serve similar needs.

To conduct the replication study we have queried the obtained dataset and charted the most interesting facts. Over all, we have been able to: (1) confirm the general trends observed in \cite{24,11}, (2) extend them to take into account the subsequent 7 years of Debian evolution history, and (3) shed some light into some of the hypotheses made at the time, thanks to the more fine-grained knowledge of source files (and in particular of their checksums) that Debsources allows. We have also found some discrepancies; for the most part they seem due to the original study considering a smaller subset of the Debian archive than we did.

Paper structure. Section 2 gives an overview of the life cycle of Debian packages and releases. Section 3 details the architecture of Debsources, while Section 4 presents our data gathering process and the resulting dataset. Section 5 discusses the results of the replication study. Before concluding, Section 6 compares Debsources with related work.

Data availability. The software, dataset, and results discussed in this paper are available, in greater detail, at \url{http://data.mancoosi.org/papers/esem2014/}.

2. DEBIAN MINING FUNDAMENTALS

Debian \cite{14} is a large and complex project. In this section we present the main notions needed for mining Debian as a collection of FOSS projects, in source code format.

The life-cycles of Debian packages and releases are depicted in Figure 1. As a distribution, Debian is essentially an intermediary between upstream authors—who release software as source code tarballs or equivalent—and final users that install the corresponding binary packages using package management tools like \texttt{apt-get} \cite{5}.

Debian package maintainers are in charge of the integration work that transforms upstream tarballs into packages. They usually work on source packages, which are bundles made of upstream tarballs (e.g., \texttt{proj.x.y.z.orig.tar.gz}), Debian-specific patches (\texttt{*.diff.gz}), and machine readable metadata (\texttt{*.dsc}). The metadata of all source packages corresponding to a Debian release are aggregated into metadata index files called \texttt{Sources}. A sample source package entry

\begin{verbatim}
Package: emacs19
Priority: standard
Section: editors
Version: 19.34-19.1
Binary: emacs19, emacs19.el
Maintainer: Mark W. Eichin <eichin@[...]> 
Architecture: any
Directory: dists/hamm/main/source/editors
Files:  
75c[...]
db5 649 emacs19_19.34-19.1.dsc  
f7[...]
d40 10875510 emacs19_19.34.orig.tar.gz  
f[...]
d[...]
d8 15233 emacs19_19.34-19.1.diff.gz
\end{verbatim}

Figure 2: sample Debian source package metadata
Table 1: Debian release information; * denotes, here and in the remainder, unreleased suites.

<table>
<thead>
<tr>
<th>ver.</th>
<th>name</th>
<th>cur. alias</th>
<th>release date</th>
<th>cycle (days)</th>
<th>archived</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>buzz</td>
<td>17/06/1996</td>
<td>n/a</td>
<td>721</td>
<td>yes</td>
</tr>
<tr>
<td>1.2</td>
<td>rex</td>
<td>12/12/1996</td>
<td>178</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td>bo</td>
<td>05/06/1997</td>
<td>175</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>2.0</td>
<td>hamm</td>
<td>24/07/1998</td>
<td>414</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>slink</td>
<td>09/03/1999</td>
<td>228</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>potato</td>
<td>15/08/2000</td>
<td>525</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>3.0</td>
<td>woody</td>
<td>19/07/2002</td>
<td>703</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>sarge</td>
<td>06/06/2005</td>
<td>1053</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>4.0</td>
<td>etch</td>
<td>08/04/2007</td>
<td>671</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>5.0</td>
<td>lenny</td>
<td>15/02/2009</td>
<td>679</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>6.0</td>
<td>squeeze</td>
<td>oldstable</td>
<td>06/02/2011</td>
<td>721</td>
<td>no</td>
</tr>
<tr>
<td>7</td>
<td>wheezy stable</td>
<td>04/05/2013</td>
<td>818</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>jessie* testing</td>
<td>tbd</td>
<td>tbd</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>n/a</td>
<td>sid* unstable</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a no</td>
<td></td>
</tr>
</tbody>
</table>

from an ancient Sources file is shown in Figure 2. Similar indexes, called Packages, exist for binary packages.

Several metadata fields are worth noting. Source packages are versioned by concatenating the upstream version, a “-” sign, and a Debian-specific version. Source packages are also organized in two-level sections: packages only containing software considered free by Debian belong to the top-level (and implicit) section main; other packages are either in the contrib or non-free top-level sections, resulting in complete sections like Section: non-free/games. Each source package gets compiled to one or several binary packages, defining the granularity at which users can install software. In Figure 2, Emacs 19 corresponds to two distinct binary packages, one for the editor itself and another one for its Elisp modules.

When ready, the maintainer uploads both source and binary packages to the development release (or “suite”) called unstable (a.k.a. sid). Since Debian supports many hardware architectures, a network of build daemons (build) fetch incoming source packages from unstable, build them for all supported architectures, and upload the resulting binary packages back to unstable.

After a semi-automatic software qualification process called migration [28], which might take several days or weeks, packages flow to the testing suite. At the end of each development cycle migrations are stopped, testing is polished, and eventually released as the new Debian stable release.

Packages are distributed to users via an ad-hoc content delivery network made of hundreds of mirrors around the world. Each mirror contains all “live” suites, i.e., the suites discussed thus far plus the former stable release (oldstable). When a new stable is released, oldstable gets stashed away to a different archive—http://archive.debian.org—which is separately mirrored and contains all historical releases.

For reference, Table 1 summarizes information about Debian suites to date, their codenames, and which suites are currently archived. We note in passing that the average development cycle of Debian stable releases is 560 days (resp. 774 over the past 12 years, since woody) with a standard deviation of 270 days (resp. 133 days).

3. ARCHITECTURE

In this paper we focus on two distinct aspects of Debsources. On the one hand Debsources is a software platform that can be deployed to gather data about the evolution of Debian and all Debian-like distributions—we present this aspect in this section. On the other hand we have set up a specific Debsources instance and used it to gather a large dataset about Debian evolution history—we discuss this aspect in the next section.

The architecture of Debsources and its data flow are depicted in Figure 3. On the back end, Debsources inputs are the mirror network (for live suites) and archive.debian.org (for archived ones). Live suites can be mirrored running periodically (e.g., via cron) the dedicated debmirror tool,1 which understands the Debian archive structure. Note that the archive format supported by debmirror is shared across all Debian-based distributions (or derivatives), e.g. Ubuntu, allowing to use Debsources on them. Archived suites require a more low-level mirroring approach (e.g., using rsync) due to the fact that the Debian archive structure has changed in incompatible ways over time.

For Debian live suites it is possible to receive “push” notifications of mirror updates—which usually happen 4 times a day—and use them to trigger debmirror runs, minimizing the update lag. To that end one needs to get in touch with a Debian mirror operator and ask for specific arrangements. Archived suite can only be mirrored in “pull” style, but they only change at each stable release, on average every 2 years. If needed, Debsources can be told to mirror only specific suites, for both live and archived suites.

After each mirror update, the Debsources updater is run. Its update logic is a simple sequence of 3 phases:

1. extraction and indexing of new packages;

2. garbage collection of disappeared packages, provided that a customizable grace period has also elapsed;

3. update of overall statistics about known packages.

Debsources storage is composed of 3 parts: the local mirror, the source packages—extracted to individual directories using the standard Debian tool dpkg-source—and a Postgres DB, whose schema is given in Figure 4. Note that throughout the paper, unless otherwise specified, we use “package” to mean “source package”. The DB contains information about package metadata, suites, and individual source files.

---

1http://packages.debian.org/sid/debmirror
A plugin system is available and accounts for Debsources flexibility. Each time the updater touches a package in the data storage (e.g., by adding or removing it), it sends a notification to all enabled plugins. Plugins can further process packages, including their metadata and all of their source code, and update the DB accordingly. Plugins can declare and use their own tables (see the starred tables in Figure 4) or use general purpose plugin tables such as metrics. In essence Debsources does the heavy lifting of maintaining a general purpose storage for Debian source code, enabling plugin authors to focus on data extraction.

To assess the usefulness of this design we have developed plugins to compute popular source code metrics: disk usage (mostly as a plugin example for developers), physical source lines of code (SLOC) using sloccount [29], user-defined “symbols” (functions, classes, types, etc.) using Exuberant Ctags, and SHA256 checksums of all source files—arguably not a metric per se, but useful to detect duplicates and refine other metrics on that basis. Note that simpler metrics like the number of source files do not need specific plugins, because Debsources already tracks individual files.

We are quite pleased with the little effort needed to implement the plugins: if we exclude boilerplate code, the most complex plugin (ctags) is ∼100 lines of Python code, most of which needed to parse ctags files. All plugins described above are part of the standard Debsources distribution.

On the front end, Debsources offers several interfaces. For final users, the Debsources web app implements a HTML + JavaScript interface with features like browsing, syntax highlighting, code annotations (via URL parameters), DB searches on metadata, and regular expression searches on the code via Debian Code Search [26]. The same features are exposed to developers via a JSON API. Additionally, scholars interested in aggregate queries can directly access the low-level Debsources DB using (Postgres) SQL.

### 4. DATASET

Debsources is not meant to be a centralized single-instance platform: multiple instances of it can be deployed and tuned to serve different distributions or data gathering needs. On the other hand there is also value in having notable Deb-sources instances and using them to maintain large datasets about the evolution of Debian. In this section we present one such instance—http://sources.debian.net or, for short, sources.d.n—and its dataset.

sources.d.n is publicly accessible and meant to track all Debian suites, both live and archived. It can be queried via the web UI and JSON API. For security reasons no public access to the underlying DB is possible, but DB dumps are available on demand. Anyone can recreate an equivalent Debsources instance by following the very same process we have used to build sources.d.n, namely:

1. deploy Debsources
2. configure it to mirror a nearby Debian mirror; optional: get in touch with mirror admins to receive push update notifications—we have obtained this for sources.d.n
3. trigger an initial update run using update-debsources
4. mirror archive.debian.org with rsync
5. inject all archived suites using suite-archive add

The process is I/O-bound and the time needed to complete it depends mostly on I/O write speed. For reference, it took us ∼5 days to inject archived suites + 8 days for the live ones ∼2 weeks—using 7.2 kRPM disks in RAID5, which is arguably a quite slow setup by today standards and certainly not one optimized for write speed. The resulting disk usage is as follows: 150 GB for the local mirror (100 GB used by live suites) + 610 GB for extracted packages + 75 GB for the DB (45 GB used by indexes on large tables) = ∼840 GB, which is quite tolerable for server-grade deployments.

sources.d.n is configured with all the plugins discussed in Section 3: disk usage, sloccount, ctags, and checksums. We haven’t thoroughly benchmarked the injection process, but a significant part of the processing time (∼40–50%) is used to compute and insert ctags in the DB.

Some figures about the major tables in sources.d.n DB are reported in Table 2. The 16 injected suites include all live suites (including small suites not discussed here like backports and -updates) and all archived suites, with the exception of Debian 1.1 buzz and 1.3 rex. The exception is because those releases did not have Sources indexes, nor .dsc files for all packages. Supporting their absence is not difficult, but requires an additional abstraction layer that is not implemented in Debsources yet. Previous studies [10, 24] have ignored the same releases, presumably for the same reasons.

Table 2: table sizes in the sources.d.n dataset

<table>
<thead>
<tr>
<th>table</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>suites_info</td>
<td>16</td>
</tr>
<tr>
<td>packages</td>
<td>28,454</td>
</tr>
<tr>
<td>suites</td>
<td>119,078</td>
</tr>
<tr>
<td>metrics* (i.e., disk usage)</td>
<td>81,582</td>
</tr>
<tr>
<td>sloccounts*</td>
<td>290,961</td>
</tr>
<tr>
<td>checkums*</td>
<td>3,495,057</td>
</tr>
<tr>
<td>ctags*</td>
<td>317,853,685</td>
</tr>
</tbody>
</table>

The exception of Debian 1.1 buzz and 1.3 rex is that those releases did not have Sources indexes, nor .dsc files for all packages.
Table 3: Debian release sizes

<table>
<thead>
<tr>
<th>suite</th>
<th>pkgs</th>
<th>files (k)</th>
<th>du (GB)</th>
<th>sloq/ctags (M)</th>
<th>pkg (k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hamm</td>
<td>1,373</td>
<td>348.4</td>
<td>4.1</td>
<td>35.1</td>
<td>25.6</td>
</tr>
<tr>
<td>sink</td>
<td>1,880</td>
<td>484.6</td>
<td>6.0</td>
<td>52.2</td>
<td>5.9</td>
</tr>
<tr>
<td>potato</td>
<td>2,962</td>
<td>686.0</td>
<td>8.6</td>
<td>69.1</td>
<td>23.3</td>
</tr>
<tr>
<td>woody</td>
<td>5,583</td>
<td>1394.5</td>
<td>18.2</td>
<td>143.3</td>
<td>25.7</td>
</tr>
<tr>
<td>sarge</td>
<td>9,050</td>
<td>2394.0</td>
<td>34.1</td>
<td>216.3</td>
<td>23.9</td>
</tr>
<tr>
<td>etch</td>
<td>10,550</td>
<td>2879.7</td>
<td>45.0</td>
<td>281.9</td>
<td>26.7</td>
</tr>
<tr>
<td>lenny</td>
<td>12,517</td>
<td>3713.9</td>
<td>61.8</td>
<td>351.0</td>
<td>28.0</td>
</tr>
<tr>
<td>squeeze</td>
<td>14,965</td>
<td>4913.2</td>
<td>89.2</td>
<td>462.5</td>
<td>30.9</td>
</tr>
<tr>
<td>wheezy</td>
<td>17,570</td>
<td>6588.1</td>
<td>125.8</td>
<td>609.2</td>
<td>34.7</td>
</tr>
<tr>
<td>jessie*</td>
<td>19,983</td>
<td>8017.1</td>
<td>157.8</td>
<td>786.7</td>
<td>39.4</td>
</tr>
<tr>
<td>sid*</td>
<td>21,232</td>
<td>9872.2</td>
<td>188.5</td>
<td>972.6</td>
<td>45.8</td>
</tr>
</tbody>
</table>

The table shows the Debian release sizes over time with data in Table 3. The table includes the suite name, number of packages, file size, disk usage, and other metrics for each release.

Average of 2.86 versions per package. The number of mappings between (versioned) packages and suites, ~120,000, is significantly higher than the number of packages due to packages occurring in multiple releases.

We index and checksum ~30 M source files, a whopping ~320 M ctags, and ~300,000 (language/package) pairs for an average of 3.56 different programming languages occurring in each (versioned) package. These are just preliminary observations that can be made on the basis of simple row counts; we will refine them in the next section.

5. MACRO-LEVEL EVOLUTION

Using the sources.d.n dataset we can replicate the findings of the former major study on macro-level software evolution [10] (reference study, or ref. study in the following). We present in this section our experiences in doing so. In addition to the general usefulness of conducting replication studies—indeed claim verification, method comparison, etc.—replicating today (2014) that study (2009) is particularly useful, because we now have data about 7 extra years (+77%, up to a total of 16 years) of evolution history pertaining to maintenance sustainability, we think it’s more appropriate to include all sections. To verify this hypothesis we have recomputed sizes using mainly only obtaining package counts closer to, but still higher than, those...

5.1 Total size

The total sizes of all considered suites are given in Table 3 and plotted over time in Figure 5. Using the sources.d.n dataset it has been easy to compute extra metrics (n. of source code files, disk usage, and ctags) in addition to those already computed in ref. study (n. of packages and SLOC).

When comparing with ref. study it is clear that we have considered more packages in each release: 300 more for hamm, up to 400 more for etch. A first potential reason is that they might have restricted their analysis to the main section of the Debian archive, whereas we have included all sections. Strictly speaking contrib and non-free are not part of Debian, but they are maintained by Debian people using Debian resources; given that several claims in software evolution pertain to maintenance sustainability, we think it’s more appropriate to include all sections. To verify this hypothesis we have recomputed sizes using main only obtaining package counts closer to, but still higher than, those...

5.2 Package size

We have studied the frequency distribution of package sizes in SLOC for all suites in the dataset. In Figure 6 we show the distributions for the two releases considered in our reference study (hamm and woody) plus the last two stable releases. Recent history confirms the observations of the ref. study: larger packages are getting larger and larger, with now 2 packages (the Linux kernel and the Chromium browser) past the 10 millions SLOC mark in the last stable release. At the same time more and more small packages enter the distribution over time, with about 50% of wheezy packages below 3,900 SLOC.

What has changed since ref. study is the relative stability back then, of the average package size—see Table 3. Post-etch the average package size has gone up gradually but considerably, from 26 kSLOC (etch) up to 34.7 kSLOC (+33%) in wheezy. It appears that the increase in the number of small packages added to the distribution is no longer enough to compensate the growth in size of large packages. A possible explanation is the emergence of more strict criteria in accepting new packages in Debian, with the effect of filtering out "non mature", and usually small, software. A more far-fetched explanation, if we take Debian as a rep...

Figure 5: Debian release sizes over time
5.3 Package maintenance

Using the sources.d.n dataset we can study package changes across releases (“package maintenance”, in the wording of ref. study) by considering in turn pairs of suites, using one of them as reference, and classifying packages in the other as: common (appearing in both suites no matter the version), removed (present in the reference but not in the other), or new (vice versa). We can furthermore identify unchanged packages (\(\subseteq\) common) as those appearing with the same version in the two suites. We have done this classification for all pairs of subsequent suites. A significant excerpt of the results is given in the upper part of Table 4.

Once again we obtain similar, but not identical results w.r.t. the reference study, which only gives common and unchanged measurements for hamm and etch. Restricting to main closes the gap almost entirely. The small number of packages that persisted unchanged from hamm to etch (148) shrank even further in jessie but is still non-zero—16 years later!—and seems to be stabilizing at around 80. Looking into those packages we find legacy, but still perfectly functional tools like netcat.

It is important to note that—even though this point is not immediately clear in ref. study—unchanged packages are not packages that have not been touched at all across releases, but only packages whose upstream version (e.g., 1.2.3) has not changed. Their Debian version might have changed, and in fact redoing the analysis using the complete package versions (e.g., 1.2.3-4) we find that unchanged packages w.r.t. hamm drop to 0 already at woody, “only” 3 releases later. This suggests that long lasting unchanged packages might have been abandoned upstream, but are still maintained in Debian via package patches, without going through the burden of replacing upstream maintainers.

To put things in perspective we have also computed the average package life, defined as the period of time between the release of the first suite in which a package appears as new (w.r.t. the previous release) and that of the first suite in which it is removed (ditto). The result is 944 days, only 20% higher than the average release duration since woody. In spite of a few long lasting unchanged entries, software in Debian seem to have a fairly high turnover.

We have also briefly looked into the percentage of common and unchanged packages w.r.t. the previous release: both values increase slightly post-etch, but now show a remarkable stability around 87% (common) and 43% (unchanged)—the ratio of change appears to be stable across releases.

An acknowledged limitation of our reference study is that, using only version information, one cannot assess the size of upstream changes: they can find out that a package in different suites went through (at least) one new upstream release, but not if that means that a single file has been changed, or rather if a large number of files have been. With file and checksum information from the sources.d.n data set we can be more precise.

In the lower part of Table 4 we compare each stable release with the preceding one (all pairs comparisons have been omitted due to space constraints). For each comparison we give the total amount of modified packages (\(\subseteq\) common \(\setminus\) unchanged), and the average percentage of files affected by the change w.r.t. the previous release. The latter ratio has been computed by comparing the sets of file checksums in the two versions: if a checksum from the previous release disappears in the new one we count that as one “file” change; the same goes for newly appearing checksums. One can certainly be more precise than this, for instance by computing the size of actual package diff-s, but that requires a dataset that includes the actual content of source files. Checksum comparison, like other fingerprinting techniques, is an interesting trade-off which arguably remains in the realm of pure metadata analyses.

The absolute number of modified packages appears to grow with the release size over time. Sarge is an exception to that rule, showing an anomalous high number of modified packages, but sarge is peculiar also in its very long development cycle, almost twice the average release duration. This suggests that the number of modified packages is also correlated with release duration. On the other hand, the average amount of modified files shows a remarkable stability post-etch, at around 60%, with larger fluctuations around that value in early releases. The percentage might seem high,
Table 4: changes between Debian releases: ‘c’ for common, ‘u’ for unchanged, and ‘m’ for modified packages

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
<th>from</th>
<th>to</th>
</tr>
</thead>
<tbody>
<tr>
<td>slink</td>
<td>potato</td>
<td>woody</td>
<td>sarge</td>
</tr>
<tr>
<td>hamm</td>
<td></td>
<td>1324c</td>
<td>1198c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>842u</td>
<td>463u</td>
</tr>
<tr>
<td>slink</td>
<td></td>
<td>1655c</td>
<td>1455c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>742u</td>
<td>384u</td>
</tr>
<tr>
<td>potato</td>
<td></td>
<td>2456c</td>
<td>2118c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>935u</td>
<td>551u</td>
</tr>
<tr>
<td>woody</td>
<td></td>
<td>4588c</td>
<td>3935c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1688u</td>
<td>1156u</td>
</tr>
<tr>
<td>sarge</td>
<td></td>
<td>7671c</td>
<td>6828c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3832u</td>
<td>2597u</td>
</tr>
<tr>
<td>etch</td>
<td></td>
<td>9230c</td>
<td>8041c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4578u</td>
<td>2906u</td>
</tr>
<tr>
<td>lenny</td>
<td></td>
<td>10530c</td>
<td>9631c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5272u</td>
<td>3676u</td>
</tr>
<tr>
<td>squeeze</td>
<td></td>
<td>1317c</td>
<td>13117c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6812u</td>
<td>5425u</td>
</tr>
<tr>
<td>wheezy</td>
<td></td>
<td>16543c</td>
<td>16543c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10132u</td>
<td>10519u</td>
</tr>
<tr>
<td>jessie*</td>
<td></td>
<td>19795c</td>
<td>19795c</td>
</tr>
</tbody>
</table>

but note that unchanged packages (i.e., 0% changes) are excluded from the count and that Debian release cycles are quite long for active upstream projects. Further by-hand investigation on selected projects have confirmed that active projects do indeed change that much over similar periods. These results seem to hint at a polarization in the evolution of individual FOSS projects, between active projects that evolve steadily and dormant, possibly feature-complete ones that cease evolving while still remaining useful.

5.4 Programming languages

The evolution of programming languages over time is also easy to study using sources.d.n. We show the most popular (in terms of SLOC) languages per release in Table 5 and their evolution over time, in both absolute and relative terms, in Figure 7. (Complete data for all suites and languages is available at http://sources.debian.net/stats/.)

This time we got significantly different numbers w.r.t. the reference study, while still confirming most of their conclusions. We wonder if an additional reason for discrepancies here might be the exclusion of Makefile, SQL, and XML from their analysis, given that sloccount excludes them by default, unless --addlangall is used. For reference, there are 5.4 MSLOC of makefile and 2.7 MSLOC of SQL in wheezy, cumulatively ∼1% of the total, unlikely to affect general trends. XML is a more significant omission though, as it is the 4th most popular language in wheezy. It is debatable whether XML should be considered a programming language, but its popularity hints at its usage for expressing program logic in declarative ways. For this reason we do not think it should be disregarded.

C is invariably the most popular language and its growth, in absolute terms, is steady; in relative terms its growth is not as fast as other languages, and most notably C++. Post-squeeze however the ratio at which C was losing ground to C++ slows down and almost entirely stops. (The increase...
in C’s popularity in jessie should probably be disregarded, due to the multiple version issue already discussed.)

Another interesting post-etch phenomenon is the decrease of shell script popularity, together with the consolidation of Perl decline. During the same period Python increases its popularity and is now the 5th most popular language. This suggests that Python is replacing Perl and shell script as a more maintainable glue code language.

Two other post-etch trends are worth noting: Lisp has almost halved its popularity and the under-representation of Java, hypothesized in ref. study, is now gone. Even though far behind C++, Java is the 3rd most popular language in recent releases, with a significant margin over the 4th, and has more than tripled its popularity since etch.

### 5.5 File size

Finally, we have computed the average file size (in SLOC) per language, and analyzed its evolution across releases. In this case the sources.d.n dataset is at loss w.r.t. our reference study, because the SLOC plugin currently does not compute the number of files per language (which needs passing --filecount to sloccount), but only SLOC counts. To compute average file sizes we have therefore divided per-language totals by the number of per-language files, computing the latter by only looking at file extensions. To do so we have adopted the same conventions used by sloccount for preliminary language classification, but we haven’t been able to further re-classify files as sloccount does, for instance on the basis of shebang lines like #!/bin/sh. This can be seen as a drawback of a metadata-only dataset, but is in fact a simple limitation of the current SLOC plugin implementation: instead of using a single table to collect per-language totals, the plugin should declare two, and use the extra one to map individual files entries to their languages as detected by sloccount. Fixing this is on our roadmap.

On the bright side, this difference opens the opportunity to methodological comparisons. Our results are shown in Table 6. Ref. study only lists average file sizes for 5 languages. Limited to those languages we note that the absolute numbers for C and Lisp are remarkably similar, suggesting that file extension detection is very accurate for those languages. Significant differences are visible for C++, where we found higher averages, probably due to the fact that the amount of C++ files is being underestimated by only looking at file extensions, likely due to extensions shared with C. Finally, we found much higher averages for shell (up to 4x), but that is more easily explained. Most shell scripts tend not to have file extensions, and have therefore been excluded from our count. Scripts that do have an extension are required by the Debian Policy to reside outside the execution $PATH. As a consequence, shipped .sh files tend to be shell libraries, used by relatively uncommon large applications written in shell script.

Despite the differences in absolute numbers we can confirm the continued stability of C, Lisp, Perl, and Java average sizes, basically unchanged over almost 20 years. The stability of C, considering its continued growth in absolute terms, is remarkable. The growth of shell script averages, already observed in ref. study, has inverted its trend and is now decreasing since etch, likely due to the already observed increase of Python popularity—whose average file size is increasing as well. A plausible general pattern for average file size growth is to increase while the corresponding language is still growing in popularity, to eventually stabilize and remain so for a long while.

### 5.6 Threats to validity

We haven’t replicated the (binary) package dependency analysis part of ref. study. We cannot replicate it exactly because currently Debsources does not retrieve Packages indexes and we consider out of scope for it to do so. On the other hand we can easily add a plugin to parse debian/control files, and extract dependencies from there. That will have the advantage of separating maintainer-defined dependencies from automatically generated ones, which arguably have a smaller impact on package maintainability.

The sources.d.n data set, due to the reasons discussed in Section 4, does not include the first 2 years of Debian release history. This has no impact on the replication study, given that our reference study didn’t consider them either. But it would still be interesting to add those years to our dataset, in order to peek into the early years of organized FOSS collections. Additionally, due to a regression in dpkgsource, we have not extracted all packages from archived suite. We have patched dpkg-source to overcome the limitation, but we are still missing a total of 12 (small) packages from archive.debian.org. We do not expect such a tiny amount to significantly impact our results.

Both sloccount and Exuberant Ctags are starting to show their age and suffer from a lack of active maintenance. During the development of Debsources we have reported various bugs against them, all related to the lack of support for “recent” languages; for instance, Scala and JavaScript are currently completely ignored by sloccount. This does not threaten the validity of the replication study, because ref. study relies on sloccount too, but it is starting to become problematic for dataset accuracy. The specific case of

6 http://bugs.debian.org/740883
JavaScript is particularly worrisome, due to its increasing popularity for server-side Node.js applications.

6. RELATED WORK

The scarcity of macro-level software evolution studies is one of the main motivations for this work. To the best of our knowledge, Barahona et al. [10] and its preliminary version [24] are the main studies in the field. We have replicated their findings and compared them with ours in Section 5.

Other works have studied the size and composition of specific releases of large FOSS distributions such as Red Hat 7.1 [29], Debian Potato [9], and Debian Sarge [2]. Our work improves over those by adding the time axis, which is fundamental in software evolution. An inconvenient of our approach is the reliance on a Debian-like archive structure. This is undoubtedly a limiting factor, but we believe it should be put in perspective considering that Debsources supports all Debian-based distributions, which account for their findings and compared them with ours in Section 5.

The Ultimate Debian Database (UDD) [20] has assembled a large dataset about Debian and some of its derivatives, and is a popular target for mining studies [30]. UDD too lacks the time axis—with the sole exception of a history table—convenience that researchers can start from. When consistently used in conjunction with FOSS platforms, that should be enough to investigate how far we can go with the evolution of large FOSS distributions, focusing on the source code of Debian and resemble the work described by Cerf [4]—who is worried about the evolution of software distribution. In spite of differences in absolute results, we have been able to confirm the general evolution trends observed back then, extend them to take into account the subsequent 7 years of history, and shed light into hypotheses made back then thanks to the fine-grained, file-level knowledge that Debsources allows.

Even though the bottom lines are the same, it is disturbing that we have not been able to either obtain identical results, or definitely ascertain the origin of the discrepancies. Empirical software engineering should be reproducible [22] and to that end we need more publicly accessible datasets that researchers can start from. When consistently used in conjunction with FOSS platforms, that should be enough to improve over the status quo.

More generally, the reproducibility issue and some of the difficulties we have encountered (e.g., the non backward compatible changes in Debian archive format and the dpkg-source regression) are instances of the more general “bit rot” problem described by Cerf [4]—who is worried about the long-term preservation of digital information, and rightfully so. We think that datasets like sources.d.n can help on both the reproducibility and information preservation front.

Several Debsources extensions are in the working. On the one hand we want to refine our ability to compute differences across releases and investigate how far we can go with fingerprinting techniques before having to compute all pairs

---

Table 6: average file size (in SLOC) per language (top-12, from left to right), based on file extension

<table>
<thead>
<tr>
<th>suite</th>
<th>ansic</th>
<th>cpp</th>
<th>java</th>
<th>xml</th>
<th>sh</th>
<th>python</th>
<th>perl</th>
<th>lisp</th>
<th>asm</th>
<th>fortran</th>
<th>cs</th>
<th>ph</th>
</tr>
</thead>
<tbody>
<tr>
<td>hamm</td>
<td>239</td>
<td>239</td>
<td>100</td>
<td>499</td>
<td>102</td>
<td>232</td>
<td>435</td>
<td>92</td>
<td>133</td>
<td>56</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>silink</td>
<td>251</td>
<td>199</td>
<td>99</td>
<td>747</td>
<td>119</td>
<td>254</td>
<td>403</td>
<td>124</td>
<td>121</td>
<td>125</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>potato</td>
<td>252</td>
<td>226</td>
<td>81</td>
<td>363</td>
<td>136</td>
<td>261</td>
<td>414</td>
<td>131</td>
<td>144</td>
<td>83</td>
<td>136</td>
<td></td>
</tr>
<tr>
<td>woody</td>
<td>255</td>
<td>303</td>
<td>89</td>
<td>230</td>
<td>141</td>
<td>255</td>
<td>434</td>
<td>245</td>
<td>154</td>
<td>163</td>
<td>121</td>
<td></td>
</tr>
<tr>
<td>sarge</td>
<td>237</td>
<td>305</td>
<td>103</td>
<td>171</td>
<td>127</td>
<td>278</td>
<td>423</td>
<td>195</td>
<td>166</td>
<td>93</td>
<td>138</td>
<td></td>
</tr>
<tr>
<td>etch</td>
<td>237</td>
<td>315</td>
<td>112</td>
<td>194</td>
<td>1875</td>
<td>269</td>
<td>383</td>
<td>229</td>
<td>167</td>
<td>119</td>
<td>179</td>
<td></td>
</tr>
<tr>
<td>lenny</td>
<td>232</td>
<td>297</td>
<td>109</td>
<td>201</td>
<td>1539</td>
<td>262</td>
<td>415</td>
<td>199</td>
<td>171</td>
<td>127</td>
<td>168</td>
<td></td>
</tr>
<tr>
<td>squeeze</td>
<td>219</td>
<td>302</td>
<td>112</td>
<td>225</td>
<td>1236</td>
<td>238</td>
<td>432</td>
<td>194</td>
<td>182</td>
<td>123</td>
<td>164</td>
<td></td>
</tr>
<tr>
<td>wheezy</td>
<td>222</td>
<td>321</td>
<td>155</td>
<td>220</td>
<td>1074</td>
<td>228</td>
<td>419</td>
<td>217</td>
<td>224</td>
<td>132</td>
<td>161</td>
<td></td>
</tr>
<tr>
<td>jessie</td>
<td>230</td>
<td>302</td>
<td>117</td>
<td>233</td>
<td>1064</td>
<td>258</td>
<td>439</td>
<td>182</td>
<td>218</td>
<td>136</td>
<td>146</td>
<td></td>
</tr>
</tbody>
</table>
diff-s. On the other hand we want to attack the ambitious goal of injecting into sources.d.n releases of as much Debian derivatives as possible, scaling up considerably the size of the ecosystem we are able to study at present. We think it is feasible to do so without switching to a version control system as data storage (which would bring its own non-trivial decisions about the adopted branching structure), but implementing instead file-level deduplication using checksums. Deduplication will also dramatically reduce the amount of resources needed to study the history of Debian development, for instance by injecting Debian sid snapshots at the desired granularity from http://snapshot.debian.org.

The largest Debsources instance to date (http://sources.debian.net) has already filled a niche in the Debian infrastructure and quickly gathered popularity due to its code browsing and search functionalities. What is more interesting from a scientific point of view is Debsources ability to turn one-shot evolution studies into live, perennial monitors of evolution traits that scholars have identified as worth of attention. We look forward to others joining us in developing Debsources plugins that allow to make more and more evolution studies perennial.

8. REFERENCES


