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 Zacchiroli
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—i.e., 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.

ESEM'14 September 18–19, 2014, Torino, Italy.

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

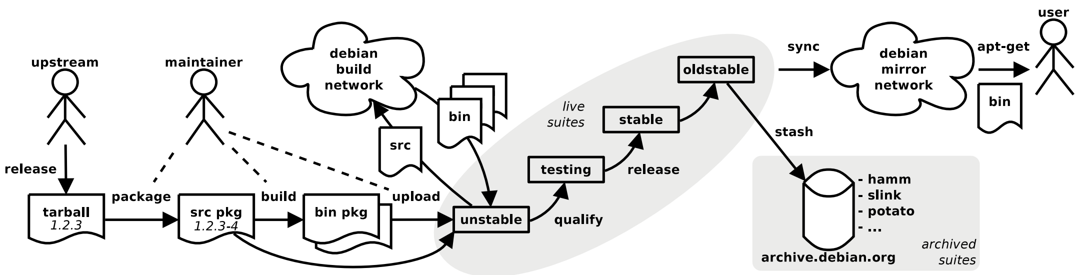


Figure 1: Life-cycle of Debian packages and releases

studies is at least in part due to the lack of suitable mining tools, storage infrastructures, and ready-to-use datasets about noteworthy software collections. With the present work we aim at contributing to fill these gaps.

Contributions. We focus on Debian,[1] 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., [28, 10, 1, 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 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 [24, 10] 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[2] 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. Overall, we have been able to: (1) confirm the general trends observed in [24, 10], (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 http://data.mancoosi.org/papers/esem2014/.

2. DEBIAN MINING FUNDAMENTALS

Debian [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 `apt-get` [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., `proj_x.y.z.orig.tar.gz`), Debian-specific patches (`*.diff.gz`), and machine readable metadata (`*.dsc`). The metadata of all source packages corresponding to a Debian release are aggregated into metadata index files called `Sources`. A sample source package entry

```
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:
 75c1[...]1db5 649 emacs19_19.34-19.1.dsc
 f715[...]84d0 10875510 emacs19_19.34.orig.tar.gz
 647d[...]1ad8 15233 emacs19_19.34-19.1.diff.gz
```

Figure 2: sample Debian source package metadata

[1] http://www.debian.org

[2] http://anonscm.debian.org/gitweb/?p=qa/debsources.git

Table 1: Debian release information; * denotes, here and in the remainder, unreleased suites.

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

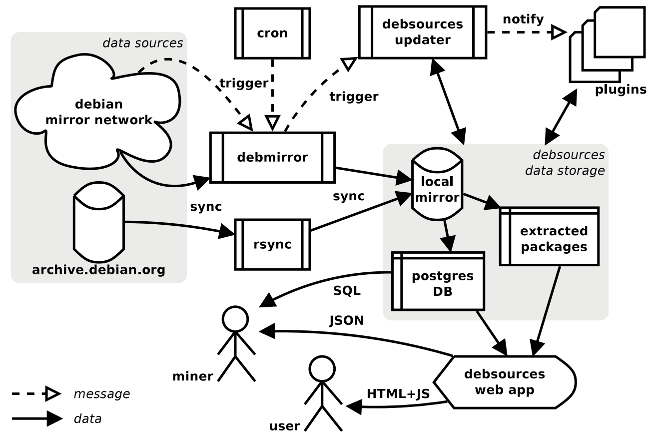


Figure 3: Debsources architecture

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 (*buildd*) 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,[3] 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.

[3] `http://packages.debian.org/sid/debmirror`

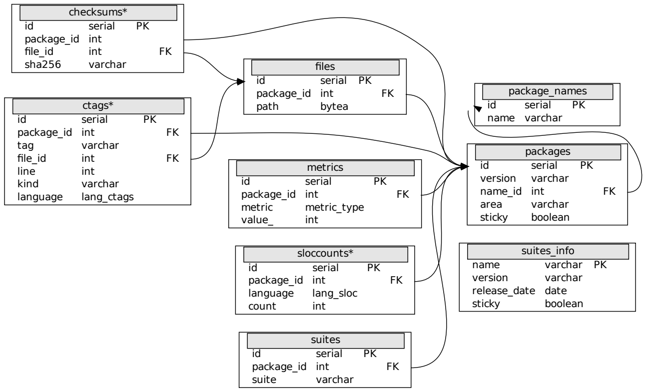


Figure 4: Debsources DB schema (excerpt); * denotes tables pertaining to plugins

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,[4] 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 Debsources 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.

[4] `http://ctags.sourceforge.net/`

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

table	rows
suites_info	16
package_names	28,454
packages	81,582
suites	119,078
metrics* (i.e., disk usage)	81,582
sloccounts*	290,961
checksums*	33,495,057
ctags*	317,853,685

`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 1.2 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.

The dataset contains ∼30,000 differently named packages, occurring in ∼80,000 distinct ⟨name,version⟩ pairs, for an

Table 3: Debian release sizes

suite	pkgs	files (k)	du (GB)	sloc (M)	ctags (M)	sloc/ pkg (k)
hamm	1,373	348.4	4.1	35.1	3.9	25.6
slink	1,880	484.6	6.0	52.2	5.9	27.7
potato	2,962	686.0	8.6	69.1	7.1	23.3
woody	5,583	1394.5	18.2	143.3	16.6	25.7
sarge	9,050	2394.0	34.1	216.3	22.9	23.9
etch	10,550	2879.7	45.0	281.9	29.0	26.7
lenny	12,517	3713.9	61.8	351.0	36.5	28.0
squeeze	14,965	4913.2	89.2	462.5	30.8	30.9
wheezy	17,570	6588.1	125.8	609.2	45.2	34.7
jessie*	19,983	8017.1	157.8	786.7	83.0	39.4
sid*	21,232	9872.2	188.5	972.6	106.5	45.8

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—independent 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 since the last release considered at the time (Debian *etch*, 2007), allowing to re-assess claims valid back then.

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[5] is that they might have restricted their analysis to the *main* section of the Debian archive, whereas we have considered 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]the URL at which the complete dataset of ref. study was available is currently broken (HTTP 404) and not available from the Internet Archive. Therefore where discrepancies exist between our findings and theirs, we have only been able to speculate about the possible causes.

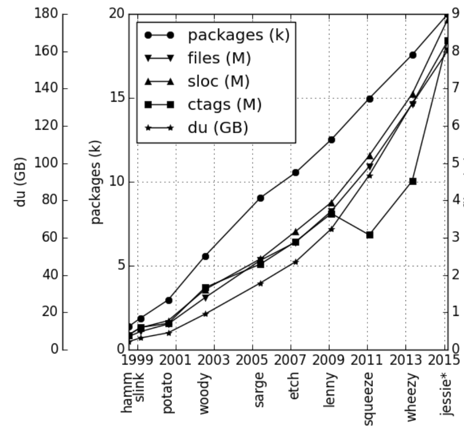


Figure 5: Debian release sizes over time

of ref. study: our dataset seems to be marginally larger—by 17 packages in *hamm*, 27 in *slink*, up to 107 in *etch*.

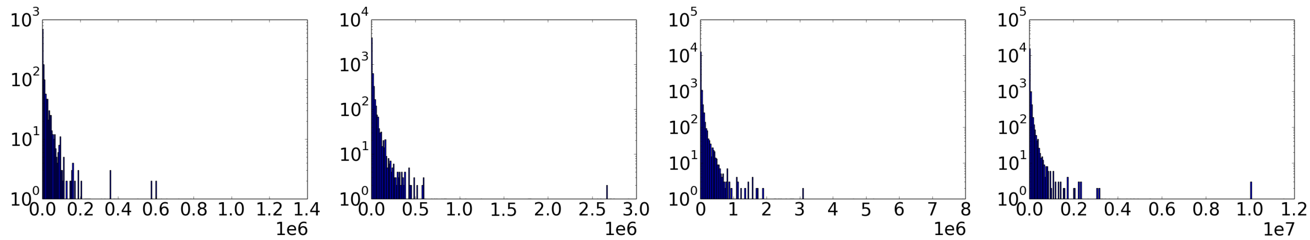
Long-term evolution trends do not seem to be affected by these differences though. Before *etch* (last release considered in ref. study) both SLOC and package counts grow linearly with time and super-linearly with releases. Interestingly, post-*etch* the growth rate has increased and is now super-linear w.r.t. time for SLOC, disk usage, and number of files; it is still linear in the number of packages though.

SLOC, disk usage, and file count metrics follow very similar patterns, confirming previous studies on metric correlation [13, 12]. Package count and ctags exhibit different patterns. The former metric, not considered in [12], might be interpreted as distribution-level refactoring, used to tame the growing complexity in the underlying upstream software, as postulated by Lehman [18]. The latter metric (ctags) exhibits a weird decrease in *squeeze* and a seemingly low value still in *wheezy*. As of now we have no good explanation for this fact; further investigation is needed.

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-



(a) hamm (b) woody (c) squeeze (d) wheezy

Figure 6: frequency (y-axis) distribution of package sizes (in SLOC, x-axis)

resentative sample of mature FOSS projects, is an increase in code contributions to large, well-established projects, at the detriment of scattered contributions over many smaller projects.

Note that data for *jessie* and *sid*, in general, should be taken with a pinch of salt, because they are under active development. In the specific case of average package sizes, we observe that those suites might temporarily contain multiple versions of very large packages such as Linux and Chromium. That might skew the averages considerably w.r.t. stable releases, where only one release per software is allowed.

Also in the case of package sizes we have obtained slightly different numbers than the reference study—in particular we observe slightly higher fluctuations in the averages—but the general picture is confirmed.

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

from \ *to*	**slink**	**potato**	**woody**	**sarge**	**etch**	**lenny**	**squeeze**	**wheezy**	**jessie***	**sid***
hamm	1324c 842u	1198c 463u	1079c 270u	958c 175u	864c 148u	782c 124u	719c 100u	670c 81u	648c 75u	663c 75u
slink		1657c 742u	1455c 384u	1281c 252u	1155c 210u	1037c 172u	941c 136u	881c 113u	852c 105u	872c 105u
potato			2456c 935u	2118c 551u	1881c 436u	1683c 352u	1497c 271u	1399c 220u	1359c 210u	1387c 211u
woody				4588c 1688u	3953c 1156u	3497c 908u	3021c 633u	2787c 520u	2680c 486u	2752c 494u
sarge					7671c 3832u	6828c 2597u	5903c 1717u	5353c 1369u	5102c 1240u	5259c 1272u
etch						9230c 4578u	8041c 2906u	7216c 2205u	6881c 1948u	7088c 2000u
lenny							10836c 5272u	9631c 3676u	9181c 3153u	9457c 3249u
squeeze								13117c 6812u	12464c 5425u	12902c 5622u
wheezy									16543c 10132u	17042c 10519u
jessie*										19795c 19593u

	from previous suite to							
	slink	**potato**	**woody**	**sarge**	**etch**	**lenny**	**squeeze**	**wheezy**
modified pkgs	556m	1305m	3127m	4462m	2879m	3287m	4129m	4453m
changed files per pkg	54.6%	64.4%	65.3%	67.5%	58.9%	59.8%	60.4%	57.2%

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*

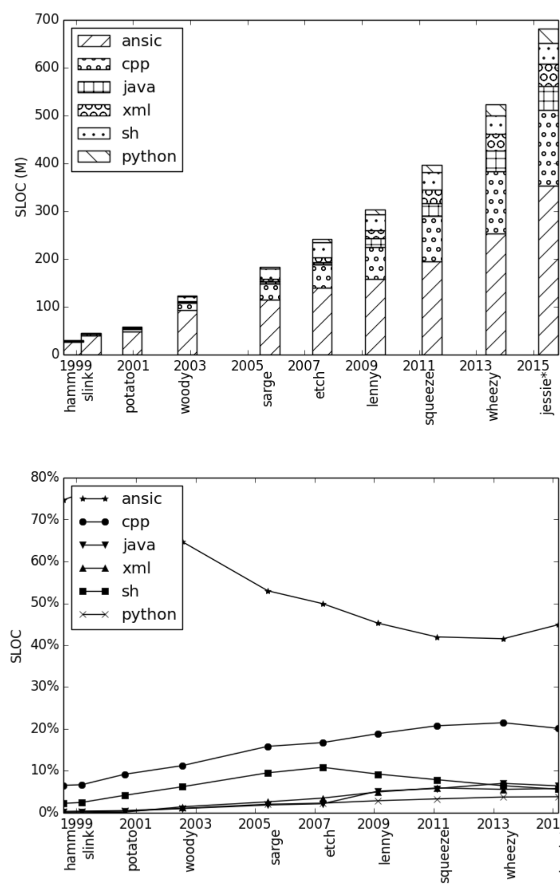


Figure 7: most popular (top-5) programming languages in Debian over time

Table 5: most popular (top-10) programming languages in Debian releases (Msloc)

release	ansic	cpp	java	xml	sh	python	perl	lisp	asm	fortran
hamm	26.2 (74.7%)	2.3 (6.5%)	0.1 (0.2%)	0.0 (0.0%)	0.8 (2.2%)	0.1 (0.3%)	0.5 (1.4%)	2.3 (6.6%)	0.4 (1.1%)	0.7 (2.0%)
slink	39.8 (76.3%)	3.5 (6.7%)	0.1 (0.3%)	0.0 (0.0%)	1.3 (2.5%)	0.2 (0.4%)	0.8 (1.5%)	2.5 (4.7%)	0.6 (1.2%)	1.0 (2.0%)
potato	47.8 (69.2%)	6.3 (9.2%)	0.3 (0.4%)	0.1 (0.2%)	2.9 (4.2%)	0.4 (0.5%)	1.3 (1.9%)	3.4 (4.9%)	0.6 (0.8%)	1.4 (2.1%)
woody	92.8 (64.8%)	16.1 (11.2%)	1.4 (1.0%)	2.0 (1.4%)	8.9 (6.2%)	1.5 (1.1%)	3.0 (2.1%)	5.1 (3.6%)	2.6 (1.8%)	2.3 (1.6%)
sarge	114.6 (53.0%)	34.3 (15.8%)	4.0 (1.8%)	5.6 (2.6%)	20.6 (9.5%)	4.4 (2.0%)	6.1 (2.8%)	6.9 (3.2%)	2.8 (1.3%)	2.9 (1.3%)
etch	140.8 (49.9%)	47.2 (16.7%)	6.1 (2.2%)	9.9 (3.5%)	30.6 (10.9%)	6.5 (2.3%)	8.0 (2.8%)	7.2 (2.5%)	4.4 (1.6%)	2.0 (0.7%)
lenny	158.9 (45.3%)	66.3 (18.9%)	18.1 (5.2%)	17.4 (5.0%)	32.4 (9.2%)	10.1 (2.9%)	9.2 (2.6%)	8.1 (2.3%)	4.1 (1.2%)	2.2 (0.6%)
squeeze	194.2 (42.0%)	96.0 (20.8%)	26.8 (5.8%)	27.4 (5.9%)	36.5 (7.9%)	15.3 (3.3%)	12.2 (2.6%)	9.5 (2.1%)	4.7 (1.0%)	2.5 (0.5%)
wheezy	253.1 (41.5%)	130.9 (21.5%)	42.8 (7.0%)	34.0 (5.6%)	39.3 (6.5%)	22.9 (3.8%)	16.1 (2.6%)	8.6 (1.4%)	7.8 (1.3%)	8.1 (1.3%)
jessie*	353.2 (44.9%)	158.6 (20.2%)	50.5 (6.4%)	45.5 (5.8%)	44.3 (5.6%)	30.1 (3.8%)	18.9 (2.4%)	11.2 (1.4%)	10.7 (1.4%)	9.1 (1.2%)

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 files 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 `dpkg-source`,[6] 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

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

suite	ansic	cpp	java	xml	sh	python	perl	lisp	asm	fortran	cs	php
hamm	239	239	100	-	499	102	232	435	92	133	56	57
slink	251	198	99	747	572	119	254	403	124	121	125	44
potato	252	226	81	363	859	136	261	414	131	144	83	136
woody	255	303	89	230	1411	137	255	434	245	154	163	121
sarge	237	305	103	171	1729	148	278	423	195	166	93	138
etch	237	315	112	194	1875	151	269	383	229	167	119	179
lenny	232	297	109	201	1539	154	262	415	199	171	127	168
squeeze	219	302	112	225	1236	152	238	433	194	182	123	164
wheezy	222	321	115	220	1074	153	228	419	217	224	132	161
jessie	230	302	117	233	1064	165	258	439	182	218	136	146

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 about 40% of all active GNU/Linux distributions and include the most popular ones (e.g., Ubuntu) [6].

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 used to store time series which, contrary to what happens in Debsources, cannot be recreated from local storage.

Numerous studies [27, 3] have investigated the evolution of individual high-profile FOSS projects (e.g., [8, 17]) and *ad hoc* sets of them (e.g., [23, 16]). Their scope is different than ours but there are synergies to be found: when investigating individual projects over long periods of time, Debsources provides a uniform interface to retrieve upstream releases as shipped by Debian; when investigating sets of projects, relying on collections like Debian can contribute to reduce project selection bias. In this respect, Debsources main limitation is granularity: it offers coherent snapshots of software releases, but not version control system (VCS) snapshots as suggested by Mockus [19]. Many studies in the literature, however, do *not* use VCSs [27].

Various studies have mined FOSS projects to detect code clones, either to enforce good engineering practices or to detect license violations, e.g., [25, 11]. When the checksum plugin is enabled, Debsources is capable of file-level clone detection and points web app users to clones. Ctags-based search can also be exploited to identify "similar" files on the basis of the symbols they define. Other fingerprinting techniques can be added by developing suitable plugins.

Boa [7] is a DSL and an infrastructure to mine large-scale collections of FOSS projects like SourceForge and GitHub. Boa's dataset is larger than Debsources (it contains SourceForge) and also more fine grained, reaching down to the VCS level, but does not correspond to curated software collections like FOSS distributions. That has both pros (it allows to peek into unsuccessful or abandoned projects) and cons: contained projects are less likely to be representative of what was popular at the time and the time horizon is more limited than with distributions as old as Debian.

FLOSSmole [15] is a collaborative collection of datasets collected by mining FOSS projects. Many datasets in there are about Debian but no one is, by far, as extensive as `sources.d.n`. We are considering submitting periodic snapshots to FLOSSmole, but the DB size makes it non-trivial.

7. CONCLUSION

We have introduced Debsources, an extensible software platform to gather data about the evolution of large FOSS collections, focusing on the source code of Debian and Debian like distributions. Scholars can use Debsources to observe decades-long evolution patterns (by injecting historical releases), as well as monitor day-by-day changes (following the evolution of live suites). To validate Debsources flexibility, we have used it to gather the largest dataset to date about Debian evolution, made it publicly available, and used it to replicate former major studies on macro-level software evolution [24, 10]. 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 `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

[1] P. Abate, J. Boender, R. Di Cosmo, and S. Zacchiroli. Strong dependencies between software components. In *ESEM*, pages 89–99, 2009.

[2] J.-J. Amor-Iglesias, J. M. González-Barahona, G. Robles-Martínez, and I. Herráiz-Tabernero. Measuring libre software using debian 3.1 (sarge) as a case study: preliminary results. *Upgrade Magazine*, Aug 2005.

[3] H. P. Breivold, M. A. Chauhan, and M. A. Babar. A systematic review of studies of open source software evolution. In *APSEC*, pages 356–365, 2010.

[4] V. G. Cerf. Avoiding "bit rot": Long-term preservation of digital information. *Proceedings of the IEEE*, 99(6):915–916, 2011.

[5] R. Di Cosmo, P. Trezentos, and S. Zacchiroli. Package upgrades in FOSS distributions: Details and challenges. In *HotSWUp*. ACM, 2008.

[6] DistroWatch distribution search. http://distrowatch.com/search.php?ostype=Linux&basedon=Debian&status=Active.

[7] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In *ICSE*, pages 422–431. IEEE / ACM, 2013.

[8] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *ICSM*, pages 131–142, 2000.

[9] J. M. González-Barahona, M. O. Perez, P. de las Heras Quirós, J. C. González, and V. M. Olivera. Counting potatoes: the size of debian 2.2. *Upgrade Magazine*, 2(6):60–66, 2001.

[10] J. M. González-Barahona, G. Robles, M. Michlmayr, J. J. Amor, and D. M. Germán. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285, 2009.

[11] A. Hemel and R. Koschke. Reverse engineering variability in source code using clone detection: A case study for linux variants of consumer electronic devices. In *WCRE*, pages 357–366. IEEE, 2012.

[12] I. Herraiz, J. M. González-Barahona, and G. Robles. Towards a theoretical model for software growth. In *MSR*. IEEE, 2007.

[13] I. Herraiz, G. Robles, and J. M. González-Barahona. Comparison between slocs and number of files as size metrics for software evolution analysis. In *CSMR*, pages 206–213. IEEE, 2006.

[14] R. Hertzog and R. Mas. *The Debian Administrator's Handbook*. Freexian SARL, 2013.

[15] J. Howison, M. Conklin, and K. Crowston. FLOSSmole: A collaborative repository for FLOSS research data and analyses. *IJITWE*, 1(3):17–26, 2006.

[16] S. Karus and H. Gall. A study of language usage evolution in open source software. In *MSR*, pages 13–22. ACM, 2011.

[17] S. Koch and G. Schneider. Effort, co-operation and co-ordination in an open source software project: GNOME. *Inf. Syst. J.*, 12(1):27–42, 2002.

[18] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[19] A. Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *MSR*, pages 11–20, 2009.

[20] L. Nussbaum and S. Zacchiroli. The ultimate debian database: Consolidating bazaar metadata for quality assurance and data mining. In *MSR*, pages 52–61. IEEE, 2010.

[21] F. P. Brooks, Jr. *The mythical man-month: essays on software engineering*. Addison-Wesley, 2 edition, 1995.

[22] G. Robles. Replicating MSR: A study of the potential replicability of papers published in the mining software repositories proceedings. In *MSR*, pages 171–180. IEEE, 2010.

[23] G. Robles, J. J. Amor, J. M. González-Barahona, and I. Herraiz. Evolution and growth in large libre software projects. In *IWPSE*, pages 165–174. IEEE, 2005.

[24] G. Robles, J. M. González-Barahona, M. Michlmayr, and J. J. Amor. Mining large software compilations over time: another perspective of software evolution. In *MSR*, pages 3–9. ACM, 2006.

[25] Y. Sasaki, T. Yamamoto, Y. Hayase, and K. Inoue. Finding file clones in FreeBSD ports collection. In *MSR*, pages 102–105. IEEE, 2010.

[26] M. Stapelberg. Debian Code Search. B.S. thesis, Hochschule Mannheim, 2012.

[27] M. M. M. Syeed, I. Hammouda, and T. Systä. Evolution of open source software projects: A systematic literature review. *JSW*, 8(11):2815–2829, 2013.

[28] J. Vouillon, M. Dogguy, and R. Di Cosmo. Easing software component repository evolution. In *ICSE*, 2014. To appear.

[29] D. A. Wheeler. More than a gigabuck: Estimating GNU/Linux's size. http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.1.03.html, 2001.

[30] J. Whitehead and T. Zimmermann, editors. *Mining Software Repositories, MSR 2010*. IEEE, 2010.