Large-scale Modeling, Analysis, and Preservation of Free and Open Source Software

Habilitation à Diriger des Recherches

(in Computer Science)

Université Paris Diderot, France

defended on 27 November 2017

by

Stefano Zacchiroli

Jury members:

Reviewers:

Ahmed Bouajjani, Full Professor, University Paris Diderot, France
Carlo Ghezzi, Full Professor, Polytechnic University of Milan, Italy
Jesus M. Gonzalez-Barahona, Associate Professor, Universidad Rey Juan Carlos, Spain

Director:

Roberto Di Cosmo, Full Professor, Inria Paris and University Paris Diderot, France

Examiners:

Jean-Bernard Stefani, Senior Researcher, Inria Grenoble-Rhône-Alpes, France
Diomidis Spinellis, Full Professor, Athens University of Economics and Business, Greece
Andreas Zeller, Full Professor, Saarland University, Germany

Contents

I Research Overview 1

1 Introduction 2
- 1.1 Contributions . . . 4
- 1.2 Reading guide . . . 7

2 Modeling FOSS packages and their relationships 9
- 2.1 Formal package modeling . . . 11
  - 2.1.1 Concrete package model . . . 12
  - 2.1.2 Abstract package model . . . 13
  - 2.1.3 On the complexity of installability . . . 14
- 2.2 Upgrade optimization . . . 15
  - 2.2.1 The Common Upgradeability Description Format . . . 16
  - 2.2.2 User preferences . . . 18
  - 2.2.3 The MISC competition . . . 19
- 2.3 Quality assurance of component repositories . . . 20
  - 2.3.1 Identifying broken packages . . . 21
  - 2.3.2 Analyzing the dependency structure of a repository . . . 21
  - 2.3.3 Predicting repository evolutions . . . 24
- 2.4 Looking back and looking forward . . . 26

3 Component modeling beyond host boundaries 29
- 3.1 A gentle introduction to Aeolus . . . 31
- 3.2 The Aeolus model . . . 35
  - 3.2.1 Resources . . . 35
  - 3.2.2 Configurations . . . 36
  - 3.2.3 Deployment actions . . . 38
  - 3.2.4 Component reconfiguration . . . 38
  - 3.2.5 Deployment runs . . . 39
  - 3.2.6 Achievability . . . 40
- 3.3 Bad news, good news: a compromise . . . 42
- 3.4 Automated cloud deployment with Aeolus . . . 43
  - 3.4.1 Architecture synthesis . . . 45
- 3.5 Aeolus toolchain internals . . . 49
  - 3.5.1 Minimizing input . . . 49
  - 3.5.2 Constraint generation . . . 50
  - 3.5.3 External solvers . . . 51
  - 3.5.4 Configuration generation . . . 51
  - 3.5.5 Synthesis soundness and completeness . . . 51

3.6 Validation . . . 52
3.6.1 Synthesis efficiency . . . 52
3.6.2 Industry adoption . . . 53

4 Delving into the source code of large FOSS collections 55
4.1 The Debsources platform . . . 56
4.1.1 Architecture . . . 56
4.1.2 Adoption . . . 59
4.2 The Debsources Dataset . . . 60
4.2.1 Exploitation ideas . . . 61
4.2.2 Availability and reproducibility . . . 62
4.2.3 Source code . . . 62
4.2.4 Metadata . . . 64
4.2.5 Dataset size . . . 66
4.3 Case study: long-term macro-level evolution . . . 67
4.3.1 Growth over time . . . 67
4.3.2 Package maintenance . . . 70
4.3.3 Programming language popularity . . . 72
4.3.4 Debian licensing over time . . . 76
4.3.5 Validation . . . 84

5 Scaling to the entire software commons 86
5.1 Requirements . . . 87
5.1.1 On the need of archiving FOSS source code . . . 87
5.1.2 Goals . . . 88
5.1.3 Core principles . . . 89
5.1.4 Use cases . . . 92
5.2 Data model . . . 95
5.2.1 Source code hosting places . . . 95
5.2.2 Software artifacts . . . 96
5.2.3 Data structure . . . 97
5.3 Architecture . . . 100
5.3.1 Listing . . . 100
5.3.2 Loading . . . 101
5.3.3 Scheduling . . . 102
5.3.4 Archive . . . 102
5.4 Current status and roadmap . . . 104
5.4.1 Listers . . . 104
5.4.2 Loaders . . . 104
5.4.3 Archive coverage . . . 104
5.4.4 Features . . . 105

6 Conclusion and future work 106
6.1 Research directions . . . 108

References 112

II Curriculum Vitae 124

7 Detailed curriculum vitae and list of publications 125

III Selected Publications 138

8 Dependency Solving: a Separate Concern in Component Evolution ... 139

9 Strong Dependencies between Software Components 172

10 Aeolus: a Component Model for the Cloud 184

11 Automated Synthesis and Deployment of Cloud Applications 223

12 Debsources: Live and Historical Views on Macro-Level Software Evolution 235

13 The Debsources Dataset: Two Decades of Free and Open Source Software 246

« If the users don’t control the program, the program controls the users. With proprietary software, there is always some entity, the “owner” of the program, that controls the program — and through it, exercises power over its users. A nonfree program is a yoke, an instrument of unjust power. »

— rms,
Free Software Is Even More Important Now

« When we’ve got these people who have practically limitless powers within a society, if they get a pass without so much as a slap on the wrist, what example does that set for the next group of officials that come into power? To push the lines a little bit further, a little bit further, a little bit further, and we’ll realize that we’re no longer citizens — we’re subjects. »

— Edward Snowden,
War on Whistleblowers

« I parigini erano sempre interessati al teatro, ma il teatro era divenuto grande quanto Parigi. I migliori oratori della Convenzione prendevano lezioni da attori consumati e la gente andava ad ascoltarli e applaudirli come se stessero sulla scena. Gli spettacoli più emozionanti erano quelli dove la gente perdeva la testa per davvero, i cannoni tuonavano e poteva capitare, da un momento all’altro, che gli spettatori si trovassero a recitare. »

— Wu Ming,
L’armata dei sonnambuli

« Ogni movimento rivoluzionario è romantico, per definizione. »

— Antonio Gramsci,
L’Ordine Nuovo, 17 gennaio 1922

For MC, TZ — the future — and the fun we are
having taking care of `tzdata`.

Part I

Research Overview

CHAPTER 1

Introduction

Free and Open Source Software (or FOSS, for short) is software that is licensed in a way that allows its users to freely use, study, modify, and redistribute it with or without modification [145]. Born in the 80s as a political movement to liberate users from the power imbalance that originates from the asymmetry between developers (who have full access to the source code of the software they write) and users (who generally don't), FOSS has taken the software industry by storm [165, 166, 109].

Today, every company who develops software, for their own needs or otherwise, *uses* FOSS; virtually every company who ships software as part of their products, also *ships* FOSS as part of them; more and more companies who use or ship FOSS, *contribute* to its development in a way or another [75]. As a result, most computer users around the world interact daily with FOSS, often without realizing it. Just to name a few examples think of the most popular operating systems for smartphones and Internet servers (Android, GNU/Linux), the leading content management system and web server (Wordpress, Apache), or the state-of-the-art in private and public "clouds" (OpenStack), not to mention programming languages implementations and development platforms that are invariably ending up being released as FOSS.

The potential impact of the *quality*, for better or worse, that such a vast body of freely licensed code has on the daily lives of billions of people around the world is beyond imaginable. It surfaces in global news only when very bad things happens, as it was the case in 2014 "thanks" to the Heartbleed vulnerability [61], because that's how news and hype work. But the fact remains that a substantial, and increasingly more so, amount of the software that runs the world is FOSS and seems to be here to remain.

The relevance of FOSS to computer science researchers and teachers, while well-known by activists who also happen to work in those fields, is much less discussed though. Those of us researchers who are also educators in programming-related classes know that FOSS has had a profound impact on course syllabuses. We now have to teach not only team collaboration, but also how to interact with third party communities to get patches reviewed and accepted; not only how to write code, but also how to find, evaluate, and *reuse* existing pieces of FOSS; not only how to release software, but also how to license it, choosing among the many FOSS licenses available; not only how to test software, but also how to nurture and efficiently manage communities of volunteers that,

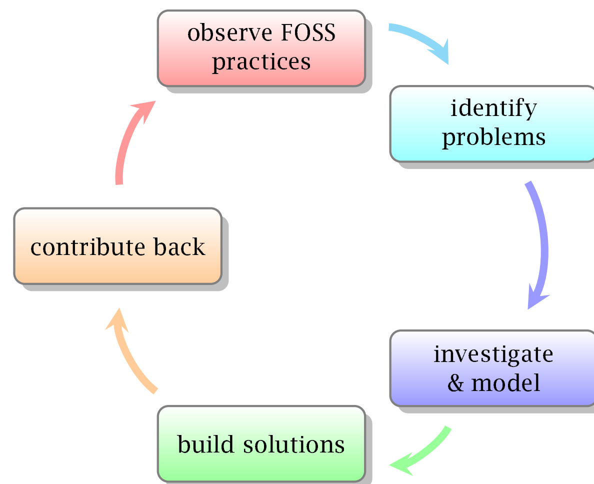


Figure 1.1: A virtuous cycle for applied software engineering in FOSS

if interested in a piece of FOSS, might come and join its development reducing maintenance costs. These changes are much more profound and durable than those induced by the rise and fall of specific *technologies*, that too often we build our teaching curricula upon.

But it is in the context of research, and in particular of applied software engineering, that FOSS has shined the most in terms of impact. On the one hand, the free availability of billions of lines of source code, together with their development histories as captured by version control systems (VCSs), is a treasure trove for empirical research. Brand new research fields were built on top of datasets coming out of FOSS, as exemplified by the Mining Software Repository case [82, 96]. On the other hand, open and collaborative development that is so typical of FOSS [131] has created a real opportunity to break the equivalent of the theatrical "fourth wall" between software developers and researchers in software engineering, as depicted in Figure 1.1.

Researchers can simply start by observing FOSS practices from the sidelines, thanks to the fact that all FOSS development outputs—releases, version control systems, build logs, bug tracking systems, code reviews, mailing list discussions, etc.—are public. They can then identify pain points and problems, either by mere observation, or by direct interaction with FOSS developers, who have shown to be very accessible to researchers that understand FOSS culture and are genuinely trying to help. This gives a wealth of information and real data that researchers can leverage to do what they do best, capture the problem in a more abstract way that it is amenable to abstract reasoning and analysis. The fact that data and information are real rather than synthetic allows to focus on the issues that actually matter and avoid "gold plating" theoretical models. While there is no silver bullet for the next step in the diagram, clear understanding of a problem often leads to natural solutions that weren't visible before—a novel algorithm, a clever optimization, or a different data model that makes the initial problem shallow.

The next, and final, two steps in the diagram are the key to a virtuous cycle between FOSS communities and applied software engineering: building real systems that show the effectiveness of the solution found to the original problem, and contributing them

back to the FOSS community making sure they are used in practice by the designated public. While stopping at the prototype level, or even before that, is totally fine for the needs of abstract research, it is a risky tactic for *applied* research. By not implementing solutions that are factually *proven* to work in the real world, we risk overlooking practical constraints that might turn a (in theory) perfectly fine solution into just another paper that (in practice) will have no visible impact on the day-by-day activities of software developers and users. Or, as Knuth famously quipped, “Beware of bugs in the above code; I have only proved it correct, not tried it.”

Furthermore, as some sort of long-term investment, it is only by actually contributing back tools and solutions to FOSS communities that researchers will show they were interested in helping out, building reputation that will help them having even easier access to the community at the next iteration.

Overall, this is a virtuous collaboration cycle that helps FOSS communities tackling hard problems that might be beyond their skill sets, and researchers in finding interesting problems that, if solved, might impact the daily lives of millions of users.

1.1 Contributions

In the context of our research work over the past decade—that this manuscript attempts to briefly summarize—both FOSS and constant collaboration with FOSS communities have been defining themes and our *modus operandi*.

Mancoosi We have first witnessed the aforedescribed virtuous cycle in the context of the EU-funded Mancoosi project, which we joined since its very beginning in 2008. The objects of study of Mancoosi were FOSS distributions, who are the most common way of distributing FOSS, in the form of *packages*, to final users. Distribution packages form very large (in the tens of thousands packages) curated collections of software components, that evolve rapidly and exhibit complex inter-component dependencies that needs to be “solved” every time a package is installed on, upgraded, or removed from user machines.

Our contribution in the context of Mancoosi are aligned along two main research directions:

distribution-wide quality assurance modeling and studying the relationships among distribution packages to various ends:

1. efficiently identify “broken” packages, i.e., packages that cannot be installed in any possible valid configuration due to their dependencies and conflicts
2. efficiently identify “sensible” packages, i.e., packages that many others depend upon—often invisibly so, without explicit dependencies declared on them by maintainers—and hence should we treated with extreme care,

Tools based on this work has since been adopted and are used daily by popular GNU/Linux distributions like Debian.

upgrade optimization to allow users to express fine-grained preferences (in the form of optimization criteria) that package managers can take into account when satisfying user requests to manipulate software packages installed on their machines.

> To this end we have developed a uniform, cross-distribution semantics of package upgrades, a corresponding document format and reference implementation, and used them as a basis to run a package upgrade competition' that tens of solvers implemented using a variety of techniques have participated into.
>
> This work has paved the way to a more modular architecture for package managers, that has since seen adoption in state-of-the-art package managers like Eclipse P2 and OPAM.

Aeolus Two main future work directions emerged at the end of Mancoosi: scaling "up" to modeling systems beyond single machine boundaries and delving "down" into the actual source code of individual software packages. The ANR[1]-funded Aeolus project, which we contributed to create and participated into since day one, picked up the first direction.

As part of Aeolus we have designed the homonymous component model, that is capable to capture not only relationships between FOSS packages within a single machine, but also the relationships among software *services* that run on different machines that communicate via local network or the Internet. The Aeolus component model is flexible enough to express the fact that service interfaces evolve over time, depending on the current readiness state of the service (e.g., "not installed", "installed but not configured", "starting up", "up and running", etc.), which wasn't the case for distribution packages.

Nonetheless, services are deployed on machines as packages. Aeolus hence preserves those mappings to avoid planning service deployment obliviously of package constraints, like conflicts, that might negatively impact it. The Aeolus component model also caters for service capacity (needed to models service load and request shaping) and dynamically spinning up and down machines, as it is common place in state-of-the-art "cloud" offerings.

The Aeolus component model has been used as the formal basis for tooling that has been built to plan optimal service deployment in "cloud" settings, with a guarantee of minimizing the needed resources, and hence deployment costs. Such tooling has seen industrial adoption by a well-known commercial distribution editor and has been used in production to formally verify the correctness of deployment plans.

Debsources Component analysis *a la* Mancoosi stops at the abstraction level of inter-package relationships. But those dependencies and conflicts generally stem from implementation characteristics, such as a source file in a package (e.g., an application) invoking a function implemented in a different package (e.g., a library). While we can detect global inconsistencies in package relationships with the Mancoosi approach, dependency correctness for individual packages can only be assessed by studying the underlying source code. Furthermore, other problematic aspects of components upgrades, such as runtime failures during deployments, can be fixed only at the source code level, and in particular by analyzing *maintainer scripts* that are shipped as part of distribution packages.

Studying and addressing these problems require dropping down to the abstraction level of source code. Unfortunately, systematic study of the source code of vast curated

[1] French national funding agency

software collections is often challenging for researchers due to the tendency of FOSS developer communities to develop *ad hoc* development tools, conventions, language elements, etc. To address this problem in the specific context of the Debian distribution we created Debsources, which is actually two things at once:

a software platform to observe live and long-term software evolution trends in the context of FOSS distributions, catering to a wide range of source code indexing and measuring needs thanks to a flexible plugin system

a dataset obtained using the Debsources platform on the Debian distribution, that covers 2 decades of FOSS evolution history for more than 3 billion lines of source code (SLOCs), as well as metadata about them such as: size metrics (lines of code, disk usage), developer-defined symbols, file-level checksums, file media types, release information, and license information.

Debsources is the largest dataset of its kind, has been adopted as a code browsing and searching platform by the Debian community, and is used by industrial consortium as a reference base for accessing Debian licensing information in a machine readable way.

Software Heritage Debsources has been a successful experiment in making a very large code base (billions SLOCs) of curated FOSS, as well as interesting metadata about it, accessible to software researchers. But it also raised a relevant question: given the intrinsic public nature of source code in FOSS, can we do more? Namely, can we collect the entire corpus of all FOSS source code ever published and systematize access to it so that software researchers can easily and reproducibly run massive-scale experiments on the entire software commons [21, 100] collected over the past decades? Our—tentative, for now—answer is that yes, we can. Proof comes in the form of the Software Heritage project, that we have co-founded with Roberto Di Cosmo and launched publicly in 2016.

The ambitious goal of Software Heritage is to collect, organize, preserve, and share with everyone who needs access to it, the entire corpus of software that has ever been published in source code form—i.e., a strict superset of the FOSS corpus—together with its full development history as captured by state-of-the-art VCSs.

In addition to massive-scale source code analysis for software engineering, Software Heritage is also meant to address the following use cases:

cultural heritage by preserving source code in the very long-term, as the sole representation of software where precious human knowledge about the technologies that run our lives resides, against increasing threats of losing some of it forever

scientific reproducibility by providing a place where source code that is relevant to scientific papers can be deposited, completing the science preservation triangle that already have good deposits for papers and dataset, but lacks one for source code

industrial software tracking by providing intrinsic and standardized identifiers for source code artifacts (individual versions of files, commits, releases, etc.), independent of third party indexes, that will allow industry players to easily track software for a variety of purposes, including security issues and the preparation of software bills of material

education by providing a stable basis on top of which a global community of educators can collaboratively curate the ultimate "source book" (spanning aspects like: the evolution of algorithm implementations across the entire history of software, real-world optimizations and data structure corresponding to pseudo-code taught in class, etc.) to train future generations of developers

1.2 Reading guide

Part I The first part of this manuscript ("Research Overview") summarizes our contributions to the state of the art of FOSS software engineering over the past decade. The presentation follows closely the organic evolution of our research interests as presented in the previous section.

Chapter 2 and Chapter 3 present our contributions in the context of the Mancoosi and Aeolus projects, respectively. Chapter 4 is dedicated to the Debsources experience. Software Heritage is described in Chapter 5. A discussion of our future research directions, all foreseen to revolve around Software Heritage, concludes the research overview part in Chapter 6.

Our research work has been published in peer reviewed journals or conference proceedings. Material from the most representative among those papers has hence been reused in preparing the relevant chapters of the research overview, putting their content in the broader context of this manuscript. Each chapter clearly indicates which papers it is based on. In the case of multi-author papers, only research work that we have either authored ourselves or contributed to in a very significant manner has been included in the research overview.

Part II The second part consists of a single chapter (Chapter 7): a detailed curriculum vitae that also includes a full list of our publications irrespectively of whether they have been covered in the research overview part or not. As it is customary nowadays for habilitation theses in France, this chapter is meant to give a broader overview of our academic work, spanning both research and non research (but still academically relevant) activities.

Part III The third and last part ("Selected Publications") of this document is a short anthology of selected papers that we have authored, and that are highlights of our research work on the various topics discussed in Part I. Some of them have been used as basis for research overview chapters, in which case they can be used as reference material to lookup omitted details, such as theorem proofs; others have not, but have been included as further readings on the discussed research topics.

The articles available in the anthology are:

[1] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Dependency solving: a separate concern in component evolution management. *Journal of Systems and Software*, 85(10):2228–2240, October 2012.

[2] Pietro Abate, Jaap Boender, Roberto Di Cosmo, and Stefano Zacchiroli. Strong dependencies between software components. In *ESEM 2009: 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 89–99, 2009.

[3] Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, and Gianluigi Zavattaro. Aeolus: a component model for the cloud. *Information and Computation*, 239:100–121, 2014.

[4] Roberto Di Cosmo, Michael Lienhardt, Ralf Treinen, Stefano Zacchiroli, Jakub Zwolakowski, Antoine Eiche, and Alexis Agahi. Automated synthesis and deployment of cloud applications. In *ASE 2014: 29th IEEE/ACM International Conference on Automated Software Engineering*, pages 211–222. ACM, 2014.

[5] Matthieu Caneill and Stefano Zacchiroli. Debsources: Live and historical views on macro-level software evolution. In *ESEM 2014: 8th International Symposium on Empirical Software Engineering and Measurement*. ACM, 2014.

[6] Matthieu Caneill, Daniel M. Germán, and Stefano Zacchiroli. The Debsources dataset: Two decades of free and open source software. *Empirical Software Engineering*, 22:1405–1437, June 2017.

CHAPTER 2

Modeling FOSS packages and their relationships

This chapter is based on the parts of [53] related to our research work as part of the Mancoosi project.

From the outset, most Free and Open Source Software (FOSS) products are installed, deployed, and maintained over time relying on so-called *distributions* [69]. The most relevant aspect of software distributions for us here is that each distribution provides a *repository*: a typically large set of software *packages* maintained and curated by distribution editors as software *components* (in the sense of [152]) that are designed to work well together.

While specific technological choices vary from distribution to distribution, many aspects, problems, and solutions are common across distributions. For instance, packages have expectations on the deployment context: they may require other packages to function properly—declaring this fact by means of *dependencies*—and may be incompatible with some other packages—declaring this fact by means of *conflicts*. Dependencies and conflicts are captured as part of *package metadata*. Figure 2.1 shows as an example of the popular Firefox web browser as a package in the Debian distribution.

A couple of observations are in order. First, note how the general form of inter-package relationships (conflicts, dependencies, etc.) is that of propositional logic formulae, having as atoms predicates on package names and their versions. Second, we have

```
Package: firefox
Version: 18.0.1-1
Depends: libc6 (>= 2.4), libgtk2.0-0 (>= 2.10), libstdc++6,
  fontconfig, procps, xulrunner-18.0, libsqlite3-0, ...
Suggests: fonts-stix | otf-stix, mozplugger,
  libgssapi-krb5-2 | libkrb53
Conflicts: mozilla-firefox (<< 1.5-1)
Provides: www-browser, gnome-www-browser
```

Figure 2.1: Package relationships for the Firefox web browser in Debian.

```
# aptitude install baobab
[...]
The following packages are BROKEN: gnome-utils
The following NEW packages will be installed: baobab [...]
The following actions will resolve these dependencies:
Remove the following packages: gnome gnome-desktop-environment libgdict-1.0-6
Install the following packages: libgnome-desktop-2 [2.22.3-2 (stable)]
Downgrade the following packages:
  gnome-utils [2.26.0-1 (now) -> 2.14.0-5 (oldstable)] [...]
0 packages upgraded, 2 newly installed, 1 downgraded,
180 to remove and 2125 not upgraded. Need to get 2442kB
of archives. After unpacking 536MB will be freed.
Do you want to continue? [Y/n/?]
```

Figure 2.2: Attempt to install the disk space monitoring utility *baobab* using the Aptitude package manager (excerpt). In response to the request, the package manager proposes to downgrade the GNOME desktop environment all together to a very old version compared to what is currently installed. As shown in [6] a trivial alternative solution exists that minimizes system changes: remove a couple of dummy "meta" packages.

various degrees of dependencies, strong ones (like "Depends") that must be satisfied as deployment preconditions and weak ones (like "Suggests" and "Recommends", the latter not shown). Finally, we also observe an indirection layer in the package namespace implemented by "Provides". Provided packages are sometimes referred to as *features*, or *virtual packages* and mean that the providing package can be used to satisfy dependencies for—or induce conflicts with—the provided name.

To maintain package assemblies, semi-automatic *package manager* applications are used to perform package installation, removal, and upgrades on user machines—the term *upgrade* is often used to refer to any combination of those actions. Package managers incorporate numerous functionalities: trusted retrieval of components from remote repositories, planning of upgrade paths in fulfillment of deployment expectations (also known as *dependency solving*), user interaction to allow for interactive tuning of upgrade plans, and the actual deployment of upgrades by removing and adding components in the right order, aborting the operation if problems are encountered at deploy-time [54].

Unfortunately, due to the sheer size of package repositories in popular FOSS distributions (in the order of tens of thousands [80]), several challenges need to be addressed to make the distribution model viable in the long run. In the following we will focus on two classes of issues and our related research work:

1. issues faced by distribution *users*, who carry the burden of maintaining their own installations functional, and
2. issues faced by *distribution editors*, who are in charge of maintaining the consistency of distribution repositories to the benefit of their users.

As motivating example of issues that are faced by users consider the seemingly simple requirement that a package manager should change as little as possible on the target machine in order to satisfy user requests. Unfortunately, as demonstrated in Figure 2.2,

```
Package: cyrus-common-2.2          Package: cyrus-common-2.4
Version: 2.4.12-1                  Version: 2.4.12-1
Depends: cyrus-common-2.4          Conflicts: cyrus-common-2.2
```

Figure 2.3: (Broken) dependencies for the Cyrus mail system in Debian.

that property is generally not guaranteed by mainstream package managers. A related issue, that we will also discuss in the following, is that of providing expressive languages that allow users of package managers to express their preferences, e.g., the demand to minimize the size occupied by packages installed on their machines.

Distribution editors, on the other hand, face the challenging task of avoiding inconsistencies in huge package archives. A paradigmatic example of inconsistency that they should avoid is that of shipping *uninstallable packages*, i.e., packages that, no matter what, cannot be installed on user machines because there is no way to satisfy their dependencies and conflicts.

Consider the (real) example involving the Cyrus mail system given in Figure 2.3. It is easy to verify that it is not possible to install the above `cyrus-common-2.2` package—a dummy package made to ease upgrades to Cyrus 2.4—out of any package repository that also contains the `cyrus-common-2.4` package shown in the example.

Even worse, it can be shown that the issue is not transitional, i.e., the team responsible for `cyrus-common-2.2` (its *maintainers*) cannot simply wait for it to go away (e.g., due to changes in *other* packages), they have to manually fix the metadata of their package so that the cause of the uninstallability goes away. The challenge here is that, while it is easy to reason on simple cases like this one, distribution editors actually need semi- or fully-automated tools able to spot this kind of quality assurance issues and point them to the most likely causes of troubles in the large maze of packages and their relationships.

2.1 Formal package modeling

Different formal treatments of packages and their relationships are needed for different purposes. Syntactic (or *concrete*) modeling captures the syntax of inter-package relationships, so that they can be treated symbolically, in a way that is close to how package maintainers reason about them. Such an approach is useful to reason about the future evolution of repositories [9, 7], as well as a basis for designing interchange formats that capture the essence of upgrade operations [154].

A more abstract package model is useful too, in order to make the modeling more independent from specific component technologies and their requirement languages. This kind of modeling is useful to recast the problem of verifying package installability as a SAT problem [114] and to capture the actual semantics of the graph of inter-package relationships [1].

2.1.1 Concrete package model

A concrete package model, originally inspired by Debian packages, has been given in [7] and further detailed in [9]. In that model packages are captured as follows:

Definition 1 (package)**.** *A* package (n, v, D, C) *consists of*

- *a* package name $n \in \mathrm{N}$,
- *a* version $v \in \mathrm{V}$,
- *a set of dependencies* $D \subseteq \wp(\mathrm{N} \times \textsc{Con})$*,*
- *a set of conflicts* $C \subseteq \mathrm{N} \times \textsc{Con}$*,*

where N is a given set of possible package names, V a set of package versions, and Con a set of syntactic constraints on them like $\top, = v, > v, \leq v, \ldots$

The intuition is that dependencies should be read as *conjunctions of disjunctions*. For example: $\{\{(p, \geq 1), (q, = 2)\}, \{(r, < 5)\}\}$ should be read as $((p \geq 1) \vee (q = 2))$ & $(r < 5)$. Starting from this intuition, the expected semantics of package constraints can be easily formalized as described below.

Notation 1. *Given a package p we write $p.n$ (resp. $p.v$, $p.D$, $p.C$) for its name (resp. version, dependencies, conflicts).*

Repositories can then be defined as package sets, with the additional constraint that name/version pairs are unambiguous package identifiers:

Definition 2 (repository)**.** *A* repository *is a set of packages, such that no two different packages carry the same name and version.*

A pair of a name and a constraint has a meaning with respect to a given repository R, the precise definition of which would depend on the formal definition of constraints and their semantics:

Notation 2. *Given a repository R, $n \in \mathrm{N}$ and $c \in \textsc{Con}$, we write $[[(n, c)]]_R$ for the set of packages in R with name n and whose version satisfies the constraint c.*

We can now capture the important notions of installation and (co-)installability:

Definition 3 (installation)**.** *Let R be a repository. An R-installation is a set of packages $I \subseteq R$ such that $\forall p, q \in I$:*

abundance *for each element $d \in p.D$ there exists $(n, c) \in d$ and a package $q \in I$ such that $q \in [[(n, c)]]_R$.*

peace *for each $(n, c) \in p.C$: $I \cap [[(n, c)]]_R = \emptyset$*

flatness *if $p \neq q$ then $p.n \neq q.n$*

```
Package:  a                Package:    b          Package:  d
Version:  1                Version:    2          Version:  3
Depends:  b (≥ 2) | d      Conflicts:  d

Package:  a                Package:    c          Package:  d
Version:  2                Version:    3          Version:  5
Depends:  c (> 1)          Depends:    d (> 3)
                           Conflicts:  d (= 5)
```

Figure 2.4: Sample Debian-like package repository

Flatness implies that only one version of a given package can be installed at a given time. Note, however, that this is a *specific* concrete package model, inspired by Debian packages. Therefore not all installation requirements listed here have equivalents in *all* component technologies. Most notably the flatness requirement varies significantly from technology to technology and, for instance, is absent from RPM packages. As discussed in [6, 9] this heterogeneity does not affect subsequent results.

Definition 4 (installability)**.** *$p \in R$ is R-installable if there exists an R-installation I with $p \in I$.*

Definition 5 (co-installability)**.** *$S \subseteq R$ is R-co-installable if there exists an R-installation I with $S \subseteq I$.*

Example 1 (package installations)**.** Consider the repository R shown in Figure 2.4. The following sets are *not* R-installations:

- R as a whole, since it is not flat;
- $\{(a,1),(c,3)\}$, since both a's and b's dependencies are not satisfied;
- $\{(a,2),(c,3),(d,5)\}$, since there is a conflict between c and d.

The following sets on the other hand are R-installations: $\{(a,1),(b,2)\}$, $\{(a,1),(d,5)\}$.

We can therefore observe that the package $(a,1)$ is R-installable, because it is contained in an R-installation. The package $(a,2)$ is not R-installable because any installation of it must also contain $(c,3)$ and consequently $(d,5)$, which will necessarily break peace. □ □

2.1.2 Abstract package model

A more abstract package model [114] has been used as basis for several advancements on issues faced by both distribution users and maintainers. The key idea is to model repositories as non mutable entities, under a *closed world assumption* stating that we know the set of all existing packages, i.e., that we will be working (at least temporarily) with respect to a fixed repository R.

Definition 6. *An* abstract repository *consists of*

- *a set of packages P,*

- *an anti-reflexive and symmetric* conflict relation $C \subseteq P \times P$,
- *a* dependency function $D \colon P \longrightarrow \wp(\wp(P))$.

The nice properties of peace, abundance, and (co-)installability can be easily recast in such a model.

The concrete and abstract models can be related. In particular, we can translate instances of the concrete model (that can easily be built from real-life package repositories) into instances of the more abstract model, preserving the installability properties. To do that, the main intuition is that (concrete) package constraints can be "expanded" to disjunctions of all (abstract) packages that satisfy them, which is a safe operation under the closed world assumption. For example, if we have a package p in versions 1, 2, and 3, then a dependency on $p \geq 2$ will become $\{\{(p,2),(p,3)\}\}$. For conflicts, we will add a conflict in the abstract model when either one of the two (concrete) packages declare a conflict on the other, or when we have two packages of the same name and different versions, to account for flatness. Formally:

Notation 3. *Let R be a repository in the concrete model. We can extend the semantics of pairs of names and constraints to sets as follows:*

$$[[\{(n_1,c_1),\ldots,(n_m,c_m)\}]]_R = [[(n_1,c_1)]]_R \cup \ldots \cup [[(n_m,c_m)]]_R$$

Definition 7 (concrete to abstract model translation)**.** *Let R be a repository in the concrete model. We define an abstract model $R_a = (P_a, D_a, C_a)$.*

- *P_a: the same packages as in R*
- *We define the dependency in the abstract model:*

$$D_a(p) = \{[[\phi]]_R \mid \phi \in p.D\}$$

- *We define conflicts in the abstract model:*

$$\begin{aligned} C_a &= \{(p_1,p_2) \mid p_1 \in [[p_2.C]]_R \vee p_2 \in [[p_1.C]]_R\} \\ &\cup \{(p_1,p_2) \mid p_1.n = p_2.n \;\&\; p_1.v \neq p_2.v\} \end{aligned}$$

2.1.3 On the complexity of installability

Now that we have rigorously established the notion of package (co-)installability, it is legitimate to wonder about the complexity of deciding these properties. Is it "easy enough" to automatically identify non-installable packages in large repositories of hundreds of thousands of packages? The main complexity result, originally established in [114] by Mancinelli et al., is not encouraging:

Theorem 1. *(Co-)installability is NP-hard (in the abstract model).*

The gist of the proof is a bidirectional mapping between boolean satisfiability (SAT) [42] and package installability. For the forward mapping, from packages to SAT, one can use one boolean variable per package (the variable will be true if and only if the corresponding package is installed), expand dependencies as implications $p \to (r_1 \vee \cdots \vee r_n)$ where r_i

Install `libc6` version `2.3.2.ds1-22` in

```
Package: libc6
Version: 2.2.5-11.8

Package: libc6
Version: 2.3.5-3

Package: libc6
Version: 2.3.2.ds1-22
Depends: libdb1-compat

Package: libdb1-compat
Version: 2.1.3-8
Depends: libc6 (>= 2.3.5-1)

Package: libdb1-compat
Version: 2.1.3-7
Depends: libc6 (>= 2.2.5-13)
```

$\Rightarrow$

$$
\begin{array}{l}
\texttt{libc6}_{2.3.2.ds1-22} \\
\wedge \\
\neg(\texttt{libc6}_{2.3.2.ds1-22} \wedge \texttt{libc6}_{2.2.5-11.8}) \\
\wedge \\
\neg(\texttt{libc6}_{2.3.2.ds1-22} \wedge \texttt{libc6}_{2.3.5-3}) \\
\wedge \\
\neg(\texttt{libc6}_{2.3.5-3} \wedge \texttt{libc6}_{2.2.5-11.8}) \\
\wedge \\
\neg(\texttt{libdb1-compat}_{2.1.3-7} \wedge \texttt{libdb1-compat}_{2.1.3-8}) \\
\wedge \\
\texttt{libc6}_{2.3.2.ds1-22} \rightarrow \\
(\texttt{libdb1-compat}_{2.1.3-7} \vee \texttt{libdb1-compat}_{2.1.3-8}) \\
\wedge \\
\texttt{libdb1-compat}_{2.1.3-7} \rightarrow \\
(\texttt{libc6}_{2.3.2.ds1-22} \vee \texttt{libc6}_{2.3.5-3}) \\
\wedge \\
\texttt{libdb1-compat}_{2.1.3-8} \rightarrow \texttt{libc6}_{2.3.5-3}
\end{array}
$$

Figure 2.5: Example: package installability as SAT instance

are all the packages satisfying the version constraints, and encode conflicts as $\neg(p \ \& \ q)$ clauses for every conflicting pair (p, q). Thanks to this mapping, we can use SAT solvers for checking the installability of packages (see Figure 2.5 and Section 2.3). The backward mapping [65], from SAT to package installability, can be established using 3-SAT instances.

Given that the proof is given for the abstract model, one might wonder to which kind of concrete models it applies. The question is particularly relevant to know whether dependency solving in the context of specific component technologies can result in corner cases of unmanageable complexity or not. Several instances of this question have been answered in [6], considering the common features of several component models such as Debian and RPM packages, OSGi [123] bundles, and Eclipse plugins [44, 36]. Here are some general results:

- Installability is NP-complete provided the component model features conflicts and disjunctive dependencies.
- Installability is in PTIME if the component model does *not* allow for conflicts (neither explicitly, nor implicitly with clauses like Eclipse's "singleton").
- Installability is in PTIME if the component model does not allow for disjunctive dependencies or features, and the repository does not contain multiple versions of packages.

2.2 Upgrade optimization

The discussed complexity results provide convincing evidence that dependency solving is difficult to get right, more than developers might imagine at first. Several authors [22, 90, 107, 159, 153, 157, 54, 93] have pointed out two main deficiencies of package managers in the area of dependency solving—incompleteness and poor expressivity—some of them have proposed various alternative solutions.

A *dependency solving problem*, as usually faced by packages managers, can be described as consisting of:

i) a repository of all available packages (also known as *package universe*);
ii) a subset of it denoting the set of currently installed packages (*package status*);
iii) a *user request* usually asking to install, upgrade, or remove some packages.

The expected output is a new package status that both is a proper installation (in the sense of Definition 3) and satisfies the user request.

Note that, due to the presence of both implicit and explicit disjunctions in the dependency language, there are usually many valid solutions for a given upgrade problem. In fact, as shown in [6], there are *exponentially* many solutions to upgrade problems in all non-trivial repositories.

A dependency solver is said to be *complete* if it is able to find a solution to an upgrade problem whenever one exists.

Given the huge amount of valid solutions to any given upgrade problem, we need languages that allow the user to express her preferences such as “favor solutions that minimize the amount of used disk space”, “favor solutions that minimize the changes to the current package status”, “do not install packages that are affected by outstanding security issues”, etc. Unfortunately, most state-of-the-art package managers are neither complete nor offer expressive user preference languages [155].

2.2.1 The Common Upgradeability Description Format

CUDF [154, 6] (*Common Upgradeability Description Format*)[1] is a document format devised to solve the issues of completeness and expressivity by inducing a synergy among package managers developers and researchers active in the various sub-fields of constraint solving.

At first glance, a CUDF document captures an instance of a dependency solving problem using a human readable syntax, as shown in Figure 2.6. CUDF is an extensible language—i.e., it allows to represent ad-hoc package properties that can then be used to express user preferences—and provides a formal semantics, based on the concrete package model of Section 2.1.1, to unambiguously determine whether a given solution is correct with respect to the original upgrade problem or not.

CUDF is agnostic with respect to specific packaging and solving technologies. Several kinds of package manager-specific upgrade problems can be translated to CUDF and then fed to solvers based on different constraint solving techniques. Figure 2.7 enumerates a number of packaging technologies and solving techniques that can be used together, relying on CUDF for data exchange.

In practice, this is achieved by instrumenting existing package managers with the ability to communicate via the CUDF format with external dependency solvers. Such an arrangement, depicted in Figure 2.8 and studied in [5, 8], allows to actually *share* dependency solvers across package managers.

[1] `http://www.mancoosi.org/cudf/`

```
preamble:
property: bugs: int = 0, suite: enum(stable,unstable) = "stable",

package: car
version: 1
depends: engine, wheel > 2, door, battery <= 13
installed: true
bugs: 183

package: bicycle
version: 7
suite: unstable

package: gasoline-engine
version: 1
depends: turbo
provides: engine
conflicts: engine, gasoline-engine
installed: true

[...]

request:
install: bicycle, gasoline-engine = 1
upgrade: door, wheel > 3
```

Figure 2.6: Sample CUDF document

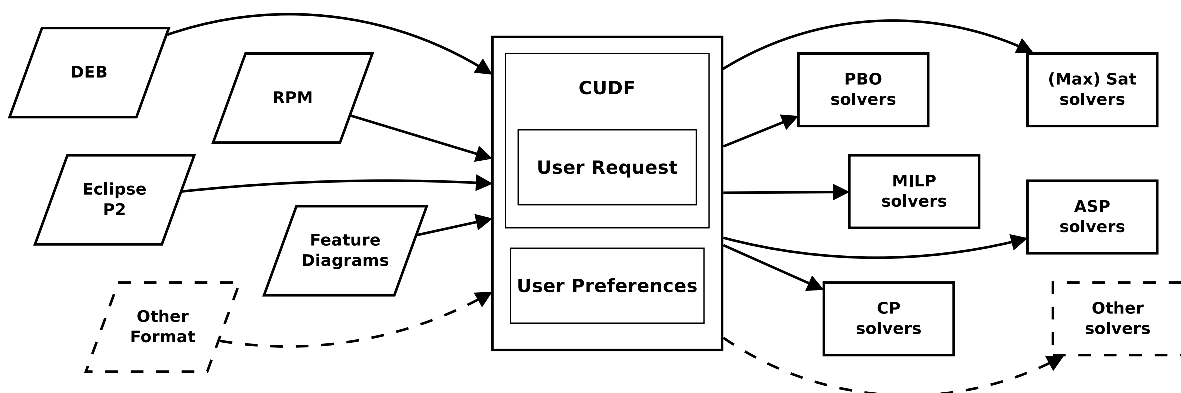


Figure 2.7: Sharing upgrade problems and solvers among distribution and research communities

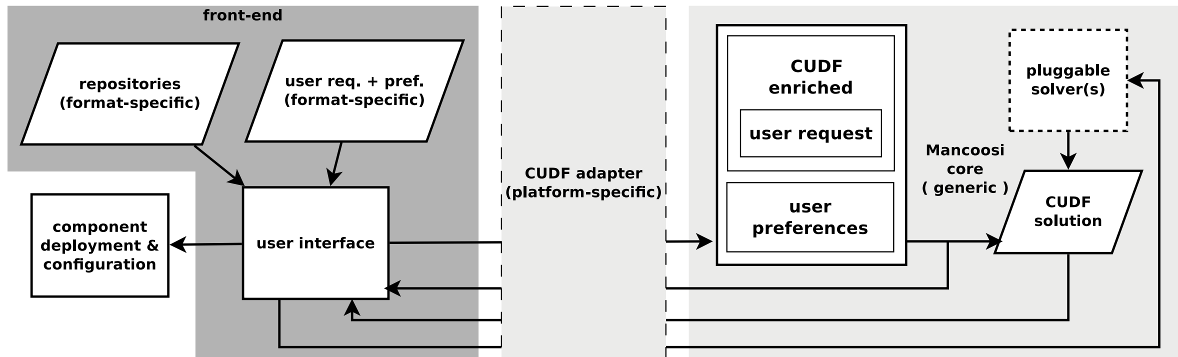


Figure 2.8: A modular package manager architecture

Adoption

We have authored a reference implementation of CUDF, which is available under the name libCUDF.[2] It is an OCaml [110] library that also comes with bindings for the C programming language. libCUDF implements the full semantics of CUDF, allowing to check that all dependencies and conflicts are respected in a given package status encoded as a CUDF document. It also allows to check that a given user request, expressed in the same format, is properly satisfied by a new package status, ideally returned by a package manager. libCUDF is released as FOSS and available as part of major FOSS distributions.

Since the release of CUDF 2.0 and libCUDF several package managers have adopted as their internal architecture the modular approach depicted Figure 2.8 and are nowadays solving dependencies using a 3rd party solver communicating with it via CUDF, using libCUDF or in-house implementations of the format. Notable examples are:

- the APT package manager used by Debian and Ubuntu, which defaults to an internal solver but can invoke external dependency solvers on demand via the `apt-cudf` wrapper;[3]
- OPAM [160],[4] the native package manager for the OCaml programming language, that has been built since the very beginning with CUDF as the abstraction layer of choice over dependency solving;
- the P2 provisioning platform for the popular Eclipse IDE and platform [36, 44], that natively speaks CUDF [106].

2.2.2 User preferences

In itself, CUDF does not mandate a specific language for expressing user preferences, but *supports* them in various ways. On the one hand, CUDF captures and exposes all relevant characteristics of upgrade problems (e.g., package and user request properties) that are needed to capture user preferences in common scenarios [155]; also, CUDF does so in an extensible way, so that properties that are specific to a given package technology

[2] https://github.com/zacchiro/cudf
[3] https://manpages.debian.org/testing/apt-cudf/apt-cudf.1.en.html
[4] https://opam.ocaml.org/

can still be captured. On the other hand, the CUDF model is rigorous, providing a solid base to give a clear and measurable semantics to user preferences, which would allow to compare solutions and decide how well they score with respect to user preferences.

Several proposals of user preference languages have been advanced. The main challenge consists in finding a middle ground between the expressivity that users desire and the capabilities of modern constraint solvers.

For the first time in [6], a flexible preference language has been proposed, based on a set of metrics that measure the *distance* between the original package status and the solution found by the dependency solver. Distance can be measured on various axes: the number of packages removed, newly installed, changed, that are not up to date (i.e., not at the latest available version), and with unsatisfied "weak" dependencies (i.e., packages that are "recommended" to be installed together with others, but not strictly required for an installation to be valid).

Those metrics can then be combined using a vocabulary of aggregation functions that are commonly supported by solvers capable of multi-criteria optimization [147], in particular lexicographic orderings and weighted sums. Using the resulting formalism it is possible to capture common user preference use cases such as those of the "paranoid user":

$$paranoid = lex(-removed, -changed)$$

The solution scoring best under this criterion is the one with the smallest number of removed packages, and then with the smallest number of changes overall (e.g., upgrade/-downgrade/install actions). A "trendy user" user case, i.e., the desire of having the most recent versions of packages, is also easy to express as:

$$trendy = lex(-removed, -notuptodate, -unsatrec, -new)$$

The set of preference combinators is bound to grow to encompass new user needs. For example, it is often the case that a single source package can produce many binary packages and that using a mix of binary packages coming from different versions of the same source package is problematic. It has been shown (in work by Di Cosmo et al.) how to implement an optimization criterion that allows to specify that some packages need to be *aligned*, for different notions of alignment [47].

2.2.3 The MISC competition

The existence of a language like CUDF allows to assemble a corpus of challenging (for existing dependency solvers) upgrade problems coming from actual users of different package managers. Using such a corpus we have established the MISC (for Mancoosi International Solver Competition),[5] which has been run yearly for three editions. The goal of the competition was to advance the state of the art of real dependency solvers, similarly to what has happened in others fields with, e.g., the SAT competition [22].

A dozen solvers have participated in the various editions, attacking CUDF-encoded upgrade problems using a wide range of constraint solving techniques. Table 2.1 shows a sample of MISC participants from the 2010 and 2011 editions.

[5] http://www.mancoosi.org/misc/

Table 2.1: Sample of MISC competition entrants, ed. 2010 and 2011

solver	author/affiliation	technique/solver
apt-pbo [157]	Trezentos / Caixa Magica	Pseudo Boolean Optimization
aspcud [72]	Matheis / University of Potsdam	Answer Set Programming
inesc [17]	Lynce et. al / INESC-ID	Max-SAT
p2cudf [107]	Le Berre and Rapicault / Univ. Artois	Pseudo Boolean Optimization / Sat4j [106]
ucl	Gutierrez et al. / Univ. Luvain	Graph constraints
unsa [117]	Michel et. al / Univ. Sophia-Antipolis	Mixed Integer Linear Programming / CPLEX [40]

Analysis of the competition results has allowed us to experimentally establish the limits of state-of-the-art solvers. In particular, they have been shown to significantly degrade their ability to (quickly) find solutions as the number of used package repositories grows, which is a fairly common use case. Each competition edition has established one or more winners, sometimes in multiple "tracks", e.g., trendy v. paranoid criteria.

Real modular package managers that follow the architecture of Figure 2.8 can then simply integrate winning solvers, or other entrants, as their dependency solver of choice. This is, in fact, what has happened in Debian, where the aspcud solver has been packaged shortly after the competition and can now be used as solver for APT.

Solvers able to handle these optimization combinators can also be used for a variety of other purposes. It is worth mentioning one of the most unusual, which is building minimum footprint virtual images for the cloud: as noticed in [129], virtual machine images often contain largely redundant package selections, wasting disk space in cloud infrastructures. Using the toolchain available in the `dose` library,[6] which is build on top of libCUDF and sits at the core of the MISC competition infrastructure, one can compute the smallest distribution containing a given set of packages. This problem has actually been used as one of the track of the 2012 edition of the MISC competition.

More details on CUDF and the MISC competition can be found in [6] and [8].

2.3 Quality assurance of component repositories

A particularly fruitful research line has attacked the problems faced by the maintainers of curated component repositories, and in particular of FOSS distributions.

A distribution maintainer controls the evolution of a distribution by regulating the flow of new packages into and the removal of packages from it. With the package count in the tens of thousands (over 50 000 in the latest Debian development branch as of this writing), there is a serious need for tools that help answering *efficiently* several different questions. Some are related to the current state of a distribution, like: "What are the packages that cannot be installed (i.e., that are *broken*) using the distribution I am releasing?", "what are the packages that block the installation of many other packages?", "what are the packages most depended upon?". Other questions concern the evolution

[6] http://www.mancoosi.org/software/

of a distribution over time, like: "what are the *broken* packages that can only be fixed by changing them (as opposed to packages they depend on)?", "what are the *future version changes* that will break the most packages in the distribution?", "are there sets of packages that were installable together in the previous release, and can no longer be installed together in the next one?".

We highlight below the most significant results obtained over the past years that allow to answer some of these questions, and led to the development of tools which have been adopted by distribution maintainers.

2.3.1 Identifying broken packages

As we have seen in Section 2.1.3, the problem of determining whether a single package is installable using packages from a given repository is NP-hard. Despite this limiting result, modern SAT solvers are able to handle easily the instances coming from real world repositories. This can be explained by observing that explicit conflicts between packages are not very frequent, even if they are crucial when they exist, and that when checking installability in a single repository one usually finds only one version per package, hence no implicit conflicts.

As a result, there is now a series of tools, all based on the original `edos-debcheck` tool developed by Jérôme Vouillon in 2006 [114], now part of the libCUDF/`dose` toolchain, that can check installability of Debian or RPM packages, as well as Eclipse plugins, very efficiently: a few seconds on commodity desktop hardware are enough to handle the 50 000 packages from the latest Debian distribution.[7]

2.3.2 Analyzing the dependency structure of a repository

Identifying non-installable packages in a repository is only the first basic analysis which is of interest for a distribution maintainer: among the large majority of packages that are installable, not all have the same importance, and not all can be installed together.

It is quite tempting, to use the number of incoming dependencies on a package as a measure of its importance. It is also tempting to analyze the dependency graph trying to identify "weak points" in it, following the tradition of studies of disease propagation in small-world networks [13]. Several studies in the literature do use explicit dependencies, or their transitive closure, to similar ends (e.g., [102, 112]).

The explicit, syntactic dependency relation $p \rightarrow q$ is however too imprecise and can be misleading in many circumstances. Intuitively, this is so because paths in the explicit dependency graph might connect packages that are incompatible, in the sense that they cannot be installed together. To avoid this issue we need to distinguish between the syntactic dependency graph and a more meaningful version of it that takes into account the actual semantics of dependencies and conflicts. This was the main motivation for introducing the notion of *strong dependency* [1] to identify the packages that are at the core of a distribution.

[7] A daily updated showcase of uninstallable Debian packages, used by the distribution for quality assurance purposes, is currently available at `https://qa.debian.org/dose/debcheck.html`. The service is currently maintained by Ralf Treinen.

Definition 8 (strong dependency)**.** *A package p strongly depends on q (written $p \Rightarrow q$) with respect to a repository R if it is not possible to install p without also installing q.*

This property is easily seen equivalent to the implication $p \rightarrow q$ in the logical theory obtained by encoding (using the abstract model of Section 2.1.2) the repository R, so in the general case this problem is co-NP-complete, as it is the dual of an installation problem, and the strong dependency graph can be huge, because it is transitive. Nevertheless, it is possible on practical instances to compute the strong dependency graph of a recent Debian distribution in a few hours on a modern multicore machine. The optimized algorithms able to do so have been discussed in [1] and are implemented in the `dose` toolchain.[8]

Once the strong dependencies graph is known, it is possible to define the *impact set* of a package, as the set of packages that strongly depend on it. Formally:

Definition 9 (impact set)**.** *Given a repository R and a package p in R, the* impact set *of p in R is the set $Is(p,R) = \{q \in R \mid q \Rightarrow p\}$.*

This is a notion of robustness, as removing p from the distribution renders uninstallable all packages in its impact set, which is not the case with direct or transitive dependencies. The size of the impact set can then be used to define a notion of *sensitivity*, i.e., how delicate "touching" a package is in a given repository, in terms of risks to (the installability of) other packages in the same repository. Formally:

Definition 10 (Sensitivity)**.** *The strong sensitivity, or simply* sensitivity*, of a package $p \in R$ is $|Is(p,R)| - 1$, i.e., the cardinality of the impact set minus 1.*[9]

Table 2.2 shows the top packages from the Debian 5.0 "Lenny" distribution with the highest sensitivity. It is easy to see that the number of direct incoming dependencies (as opposed to strong dependencies) is not a good predictor of sensitivity, while the number of transitive incoming dependencies is generally an over-approximation.

Figure 2.9 gives further evidence of this, by plotting sensitivity and an equivalent measure (*direct* sensitivity) computed using direct dependencies instead of strong ones. In most cases, strong sensitivity is higher than direct sensitivity, yet close: 82.9% of the packages fall in a standard deviation interval from the mean of the difference between strong and direct sensitivity; the next percentile ranks are 97.4% for two standard deviations, and 99.8% for three. The remaining cases allow for important exceptions of packages with very high strong sensitivity and very low direct sensitivity. Such exceptions are extremely relevant: metrics built on direct sensitivity only would totally overlook packages with a huge potential impact.

In the list of Table 2.2, an experiecned maintainer will recognize a cluster of interrelated packages: `gcc-4.3-base`, `libgcc1` and `libc6` are all essential components of the C library, and they have similar sized impact sets. In the general case, though, as shown

[8] http://www.mancoosi.org/software/

[9] The -1 accounts for the fact that the impact set of a package always contains itself. This way we ensure that sensitivity 0 preserves the intuitive meaning of "no package potentially affected".

Table 2.2: Top sensitive packages in Debian 5.0 “Lenny”

#	package	deps	**strong deps**	closure
1	`gcc-4.3-base`	43	**20128**	20132
2	`libgcc1`	3011	**20126**	20130
3	`libc6`	10442	**20126**	20130
4	`libstdc++6`	2786	**14964**	15259
5	`libselinux1`	50	**14121**	14634
6	`lzma`	4	**13534**	13990
7	`libattr1`	110	**13489**	14024
8	`libacl1`	113	**13467**	14003
9	`coreutils`	17	**13454**	13991
10	`dpkg`	55	**13450**	13987
11	`perl-base`	299	**13310**	13959
12	`debconf`	1512	**11387**	12083
13	`libncurses5`	572	**11017**	13466
14	`zlib1g`	1640	**10945**	13734
15	`libdb4.6`	103	**9640**	13991
		...		

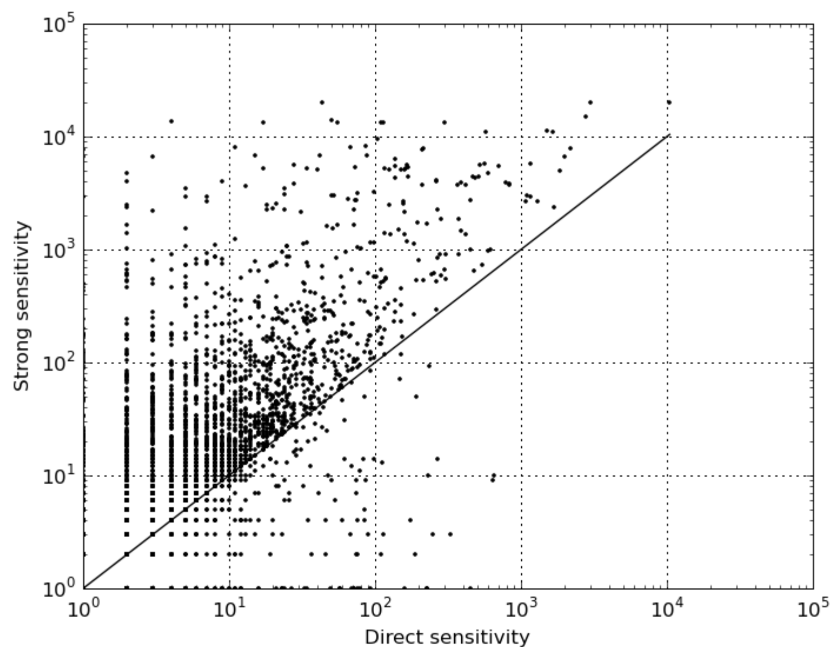


Figure 2.9: Correlation between strong and direct sensitivity in Debian 5.0 “Lenny”

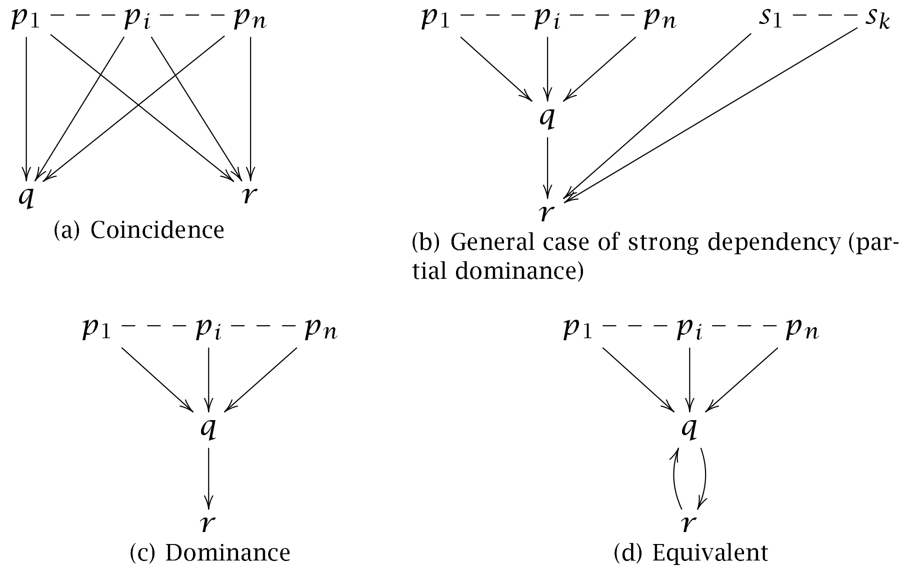


Figure 2.10: Significant configurations in the strong dependency graph

by the sample configurations drawn in Figure 2.10, two packages having similar sensitivity need not be correlated. To identify the packages that are correlated, and identify the most relevant ones among them, one can define, on top of the strong dependency graph, a dominance relation similar to the one used in traditional flow graphs [108], as follows:

Definition 11 (strong dominance). *We say that q strongly dominates r if:*

- *q strongly depends on r, and*
- *every package that strongly depends on r also strongly depends on q.*

Intuitively, in a strong dominance configuration, that looks like Figure 2.10(c), the strong dependency on r of packages in its impact set is "explained by" their strong dependency on q.

Strong dominance can be computed efficiently [25] and properly identifies many relevant clusters of packages related by strong dependencies like the `libc6` one. Similarly, it is possible to capture and efficiently compute *partial* dominance situations like that shown in Figure 2.10(b). For more details see [1].

2.3.3 Predicting repository evolutions

Some aspects of the quality assessment in FOSS distributions are best modeled by using the notion of *futures* [2, 7] of a package repository. This allows to investigate under which conditions a potential future problem may occur, or what changes to a repository are necessary to make a currently occurring problem go away. This analysis can give package maintainers important hints about how problems may be solved, or how future problems may be avoided. The precise definition of these properties relies on the definition of the possible future of a repository:

```
Package: bar
Version: 2.3

Package: baz
Version: 2.5
Conflicts: bar (> 2.4)
```

```
Package: foo
Version: 1
Depends: (baz (=2.5) | bar (=2.3)),
         (baz (<2.3) | bar (>2.6))
```

Figure 2.11: Package `foo` in version 1 is outdated

Definition 12 (future)**.** *A repository F is a* future *of a repository R if the following two properties hold:*

uniqueness *$R \cup F$ is a repository; this ensures that if F contains a package p with same version and name as a package q already present in R, then $p = q$;*

monotonicity *For all $p \in R$ and $q \in F$: if $p.n = q.n$ then $p.v \leq q.v$.*

In other words, when going from the current repository to some future of it one may upgrade current versions of packages to newer versions, but not downgrade them to older versions (monotonicity). One is not allowed to change the meta-data of a package without increasing its version number (uniqueness), but besides this the upgrade may modify the meta-data of a package in any possible way, and may even remove a package completely from the repository, or introduce new packages.

This notion models all the changes that are possible in the maintenance process usually used by distribution editors, even if the extreme case of a complete change of meta-data allowed in this model is quite rare in practice. Note that the notion of future is not transitive as one might remove a package and then reintroduce it later with a lower version number.

The first property related to futures that we are interested in is the following one:

Definition 13 (outdated)**.** *Let R be a repository. A package $p \in R$ is* outdated *in R if p is not installable in any future F of R.*

That is, p is outdated in R if it is not installable (since R is itself one of its futures) and if it has to be upgraded to make it ever installable again. In other words, the only way to make p installable is to upload a fixed version of the package, since no modification to *other* packages than p can make p installable. This information is useful for quality assurance purposes, as it pinpoints packages where action is required. An example of an outdated package is given in Figure 2.11.

Definition 14 (challenges)**.** *Let R be a repository, $p, q \in R$, and q installable in R. The pair $(p.n, v)$, where $v > p.v$,* challenges *q if q is not installable in any future F which is obtained by upgrading p to version v.*

Intuitively $(p.n, v)$ challenges q, when upgrading p to a new version v without touching any other package makes q not installable. This permits to pinpoint critical *future*

```
Package: foo
Version: 1.0
Depends: bar (<= 3.0) | bar (>= 5.0)

Package: bar
Version: 1.0

Package: baz
Version: 1.0
Depends: foo (>= 1.0)
```

Figure 2.12: Package `bar` challenges package `foo` for versions in the interval $]3.0, 5.0[$.

upgrades that challenge many packages and that might therefore need special attention before being pushed to the repository. An example is given in Figure 2.12.

The problem in deciding these properties is that any repository has an infinite number of possible futures. The two properties we are interested in belong to the class of so-called *straight* properties. For this class of properties it is in fact sufficient to look at a finite set of futures only which cover all of the problems that may occur in any future. One can show [7] that it is sufficient to look at futures where no package has been removed and new packages have been introduced only when their name was already mentioned in R, and where all new versions of packages have no conflicts and no dependencies. For any package there is an infinite space of all future version numbers, however, there is only a finite number of equivalence classes of these with respect to observational equivalence where the observations are the constraints on versions numbers used in R.

In reality, the definition of a future is more involved than the one given in Definition 12. In almost all distributions, packages are in fact not uploaded independently from each other but are updated together with all other packages stemming from the same *source package*. The complete definition of a future hence also takes into account a notion of *clusters* of packages, which are in our case formed by all binary packages stemming from the same source. Definition 14 has to be adapted accordingly, by allowing for all packages in the same cluster as p to be upgraded. The full version of the algorithms in presence of package clusters, together with their proof of soundness, can be found in [7].

The top challenging upgrades in Debian 5.0 “Lenny” are shown in Table 2.3. The tools used to efficiently compute outdated and challenging upgrades have been integrated into the `dose` suite and can be found in the `dose-extra` package on Debian-based distributions. Regularly updated reports on outdated Debian packages are available as part of the distribution quality assurance infrastructure.[10]

2.4 Looking back and looking forward

Mancoosi has been an ambitious research project that targeted and practically solved real issues faced by distribution users and editors, producing tools that have been adopted in

[10] https://qa.debian.org/dose/outdated.html. The service is currently maintained by Ralf Treinen.

Table 2.3: Top 13 challenging upgrades in Debian 5.0 “Lenny”

Source	**Version**	**Target Version**	**Breaks**
python-defaults	2.5.2-3	≥ 3	1079
python-defaults	2.5.2-3	$2.6 \leq . < 3$	1075
e2fsprogs	1.41.3-1	any	139
ghc6	6.8.2dfsg1-1	$\geq$ 6.8.2+	136
libio-compress-base-perl	2.012-1	$\geq$ 2.012.	80
libcompress-raw-zlib-perl	2.012-1	$\geq$ 2.012.	80
libio-compress-zlib-perl	2.012-1	$\geq$ 2.012.	79
icedove	2.0.0.19-1	$>$ 2.1-0	78
iceweasel	3.0.6-1	$>$ 3.1	70
haskell-mtl	1.1.0.0-2	$\geq$ 1.1.0.0+	48
sip4-qt3	4.7.6-1	$>$ 4.8	47
ghc6	6.8.2dfsg1-1	6.8.2dfsg1+ $\leq . <$ 6.8.2+	36
haskell-parsec	2.1.0.0-2	$\geq$ 2.1.0.0+	29

production by major FOSS vendors. What made this possible is the thorough application of the virtuous cycle between FOSS and research communities discussed in Chapter 1. In the context of Mancoosi, specific instances of using that cycle can be observed in the analysis of broken and outdated/challenging packages, in the CUDF format, and in the MISC competition that resulted in new solver technologies adopted by a variety of package managers.

In addition to the research topics presented in this chapter, Mancoosi also attacked the issue of runtime failures that might happen when deploying package upgrades on user machines—a phase of package upgrades that follows dependency solving [54]. The approach used to attack this problem has been similar to what we have seen: tens of thousands *maintainer scripts* (the shell scripts executed during package upgrades to finalize package configuration during and after deployments) have been analyzed, modeled using metamodeling techniques, in order to devise an upgrade simulator that can predict a specific class of runtime upgrade failures. The gory details of this part of our research work has been described in [45].

This latter part of the Mancoosi journey has not seen wide adoption yet, due to the fact that shell script as a language is very resistant to static analysis (in turn due to their inherent dynamic properties). As a result the class of detectable issues with the previous approach was deemed too small to be of interest for distribution maintainers. As of this writing, a novel approach is being tried as part of the ANR-funded Colis project[11] who has picked up again this challenge, and delivered promising results already [91, 92] as well as a concrete possibility of devising more general static analysis tools for shell script and dynamic languages. On this front, time will tell.

Other scientific challenges in the field of FOSS components deployment and analysis were identified at the end of Mancoosi. First and foremost we can observe that upgrade deployment in common FOSS deployments isn't any longer affected only by *in-host* con-

[11] http://colis.irif.univ-paris-diderot.fr/

straints as captured by inter-package relationships. It is also affected, and more and more so, by *extra-host* constraints as expressed by, often implicit, dependencies and conflicts among *services* running on different machines that are meant to work together over the network. Examples of this are web applications running behind a web accelerators or proxies and backed by database systems; all these systems need to be configured to work well together and upgrades should be planned and timed carefully to avoid temporary (or worse) downtimes.

The Aeolus project, discussed in the next chapter, picked up this very challenge by introducing a component model, and tooling for it, suitable to model FOSS packages, services, and upgrades in current networked and "cloud" contexts.

Chapter 3

Component modeling beyond host boundaries

This chapter is based on [52, 50].

As we have seen, FOSS practices rely on distributions to convey and deploy software components from software vendors to final users of individual machines. In-production services are rarely confined within the boundaries of a single machine though. To be modular, resilient, and scalable any "serious", say, Internet-based service will need a plethora of different software components—e.g., load balancers, proxies, firewalls, databases, application servers, etc.—and will generally run them on multiple machines that are interconnected via public and private networks. Furthermore, with the advent of "*cloud computing*" [83] both machines and software services running on them will be dynamically provisioned and disposed of, to withstand load peaks or simply to deploy multiple instances of the entire infrastructure on demand. Reaping the benefits of such complex application architecture is not an easy task: even when the infrastructure costs fall dramatically, the complexity of designing and maintaining distributed scalable software systems remains a serious challenge, when it does not outright increase.

Attempts are being made both in industry and in the research world to model and tame such complexity. On the industry side, a wealth of initiatives offer different kinds of solutions for isolated aspects of the problem. Configuration management tools like Puppet [98] or Chef [122] allow to automate the configuration of software components, relying on a set of descriptions stored in a central server. CloudFoundry [37] allows to select, connect, and push to a cloud some predefined services (databases, message buses, proxies, etc.), that can be used as building blocks for writing applications using one of the supported frameworks. Juju [95] tries to extend the basic concepts of package managers (in the distribution sense we discussed in the previous chapter) to automate service upgrades (as opposed to only distribution packages).

On the academic side, several teams have worked on the problems posed by the complexity of designing network- and cloud-based applications. The Fractal component model [29] focuses on expressivity and flexibility: it provides a general notion of component assembly that can be used to describe concisely, and independently of the programming language, complex software systems. Building on Fractal, FraSCAti [143] provides a middleware that can be used to deploy applications in the cloud. ConfSolve [84] on

the other hand aims at helping the application designer with some of the decisions to be made, and more specifically to optimally allocate virtual machines to concrete servers.

In all the these approaches the goal is to allow the user (i.e., the application designer) to assemble a working system out of components that have been specifically designed or adapted to work together. The actual component selection (which web server should I use? which SQL database? which load balancer?) and interconnection (which front-end should I connect to which back-end, in order to avoid bottlenecks?) are the responsibility of the user. And if some reconfiguration needs to happen, it is either obtained by reassembling the system manually, or by writing specific code that is left for the user to write. We argue that to make further progress in taming the complexity of sophisticated cloud applications, two major concerns must be taken into account: expressivity and automation.

Expressivity We need component models that are expressive enough to capture all the characteristics of software components that are relevant for designing distributed, scalable applications which are now commonplace in the cloud. Some of those characteristics (discussed in more details in Section 3.1 later on) are:

dependencies and conflicts similar to what we have seen for distribution packages, but with the possibility of reaching across machine boundaries. A service running on a given host should be able to depend on (or conflict with) services running elsewhere.

non-functional requirements e.g., if a component depends on others, how many of those would be needed to guarantee the desired level of fault-tolerance and/or load-balancing? Similarly, if a component offers functionalities to other, how many of them it can reasonably satisfy before needing to be replicated?

statefulness distributed-/cloud-components have complex activation protocols, making their contextual requirements (dependencies, conflicts, etc.) vary over time, e.g., it might be enough to *install* a given component to be able to install another one, but the requirements to bring it in production might be different

Automation While expressivity is important, solving the challenge of designing and maintaining a networked or cloud-based application also requires automation. When the number of components grows, or the need to reconfigure entire systems occurs more frequently, it is essential to be able to specify at a certain level of abstraction a particular target configuration of the distributed software system we want to realize, and to develop tools that provide a set of possible evolution paths leading from the current system configuration to one that corresponds to such a user request.

In Sections 3.1 through 3.3 we lay the theoretical foundations of such an automated approach—in the form of a formal component model called Aeolus—for the complex situation that arises when one needs to: (re-)configure not a single machine, but a variety of possibly “elastic” clusters of heterogeneous machines, living in different domains and offering interconnected services that need to be stopped, modified, and restarted in a specific order for the reconfiguration to be successful. Later, in Sections 3.4 through 3.6 we present some of the tooling that have been built, as part of the Aeolus project, on top

```
Package: wordpress
Version: 3.0.5+dfsg-0+squeeze1
Depends: httpd, mysql-client, php5, php5-mysql,
  libphp-phpmailer (>= 1.73-4), [...]

Package: mysql-server-5.5
Source: mysql-5.5
Version: 5.5.17-4
Provides: mysql-server, virtual-mysql-server
Depends: libc6 (>= 2.12), zlib1g (>= 1:1.1.4), debconf, [...]

Package: apache2
Version: 2.4.1-2
Maintainer: Debian Apache Maintainers <debian-apache@...>
Depends: lsb-base, procps, perl, mime-support,
  apache2-bin (= 2.4.1-2), apache2-data (= 2.4.1-2)
Conflicts: apache2.2-common
Provides: httpd
Description: Apache HTTP Server
```

Figure 3.1: Debian package metadata for Wordpress, MySQL and the Apache web server (excerpt)

of the homonymous component model to automate deployment of complex distributed applications in the cloud and discuss its adoption.

3.1 A gentle introduction to Aeolus

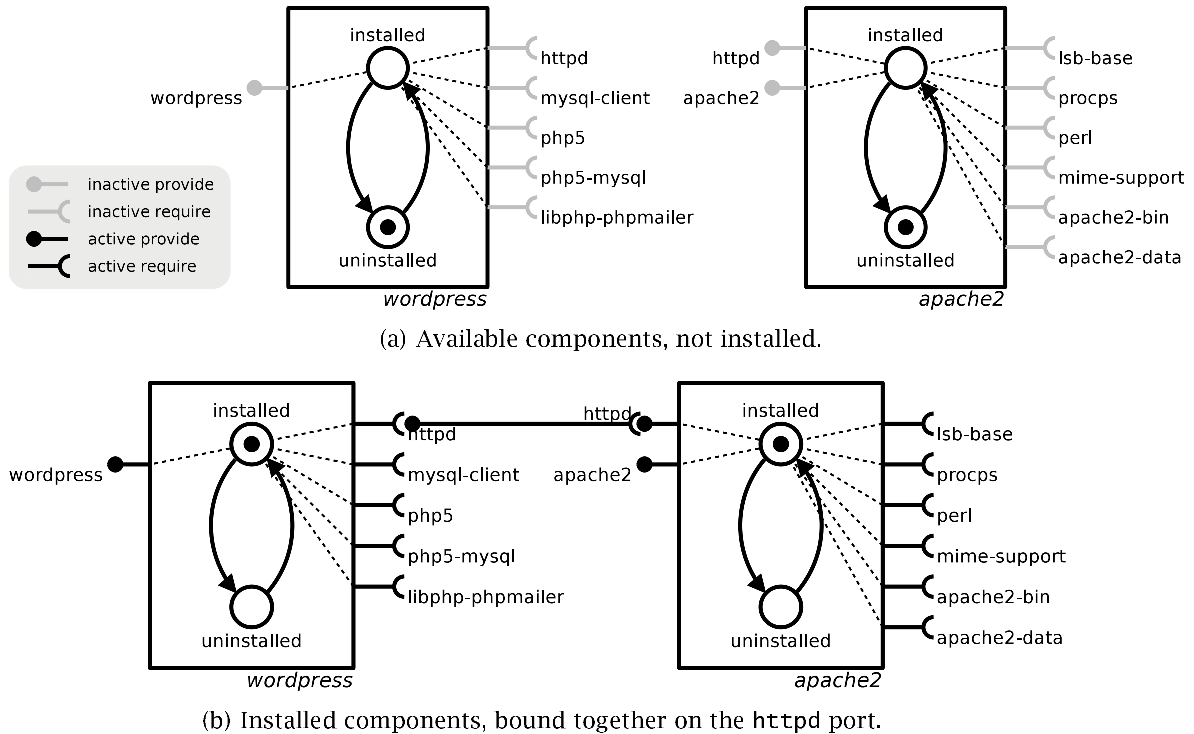
We introduce the key features of the Aeolus component model by eliciting them, step-by-step, from the analysis of realistic scenarii. As a running example, we consider several deployment use cases for Wordpress, a popular blogging platform that requires several software services to operate, the main ones being a Web server and a SQL database. We present the use cases in order of increasing complexity ranging from the simplest one, where everything run on a single machine, to more complex ones where the whole appliance runs on a platform-as-a-service (IaaS) cloud.

A *formal* definition of the Aeolus component model will be given in Section 3.2.

Use case 1 — package installation

Before considering the services that a machine is offering to others (locally or over the network), we need to model the *software installation* on the machine itself, so we will see how to model the three main components needed by Wordpress, as far as their installation is concerned. We will build on the assumption that the machine runs a FOSS distribution. For instance, Debian has packages for Wordpress, Apache2 and MySQL equipped with the metadata shown in Figure 3.1.

To model a software package, at this level of abstraction—but at the same time generalize over the boolean model of the previous chapter—we will use a simple state machine



(a) Available components, not installed.

(b) Installed components, bound together on the `httpd` port.

Figure 3.2: Graphical notation for packages and dependencies

to capture the package life cycle, with requirements and provides associated to each state.

The ingredients of this model are very simple: a set of states Q, an initial state q_0, a transition function T from states to states, a set $\mathbf{R}$ of requirements, a set $\mathbf{P}$ of provides, and a function D that maps states to the requirements and provides that are *active* at that state. We call *component type* any such tuple $\langle Q, q_0, T, \langle \mathbf{P}, \mathbf{R} \rangle, D \rangle$. We are essentially recasting in a state machine setting the boolean model of Chapter 2; the reason we are doing that is to be able to obtain a *uniform* model that, using a single formalism (state machines) can capture both package installation and service activation protocols.

A system *configuration* is then built out of a collection of components that are instances of component types, with its current state, and a set of connections between requirements and provides of the different components. Connections indicate which provide is fulfilling the need of which requirement. A configuration is *correct* if all the requires which are active are satisfied by active provides.

Graphical notation A straightforward graphical notation can capture all these pieces of information together: Figure 3.2 shows systems built using the components from Figure 3.1 (only modelling the dependency on `httpd` underlined in the metadata, for the sake of conciseness). In Figure 3.2(a) the packages are available but not installed yet. In Figure 3.2(b) the Wordpress package is in the installed state and activates the requirement on `httpd`; Apache2 is also in the installed state, so the `httpd` provide is active and is used

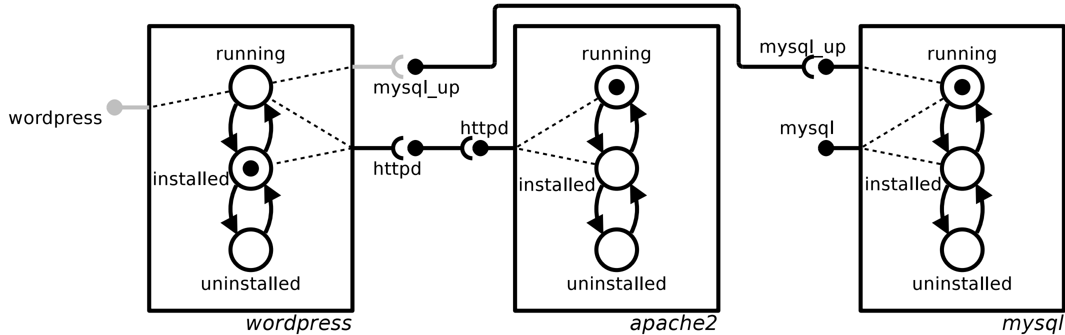


Figure 3.3: A graphical description of the basic model of services and packages.

to satisfy the requirement, fact which is visualized by the *binding* connecting together the two *ports* named `httpd`.

Packages may of course also conflict with other packages which are incompatible with them: we will show later how to model conflicts.

Use case 2 — services and packages

As we have seen, installing software on a single machine is a process that can already be automated using *package managers*: on Debian for instance, you only need to have an installed Apache server to be able to install Wordpress. But bringing it *in production* requires to tune and activate the associated service, which is more tricky and less automated: the system administrator will need to edit configuration files so that Wordpress knows the network addresses of an accessible MySQL instance.

The ingredients we have seen up to now in our model are sufficient to capture the dependencies among services, as shown in Figure 3.3. There we have added to each package an extra state (`running`) corresponding to the activation of the associated service, and the requirement on `mysql_up` of the `running` state of Wordpress captures the fact that Wordpress cannot be started before MySQL is running. In this case, the bindings really correspond to a piece of configuration information, i.e., where to find a suitable MySQL instance.

Notice how this model does not impose any particular way of modelling the relations between packages and services. Instead of using a single component with an installed and a running state, we can simply model services and packages as different components, and relate them via dependencies.

Use case 3 — redundancy, capacity planning, and conflicts

Services often need to be deployed on different machines to reduce the risk of failure or to increase the load they can withstand by the means of load-balancing. To properly design such scalable architectures system administrators might want, for instance, to indicate that a MySQL instance can only support a certain number of connected Wordpress instances. Symmetrically, a Wordpress hosting service may want to expose a reverse web

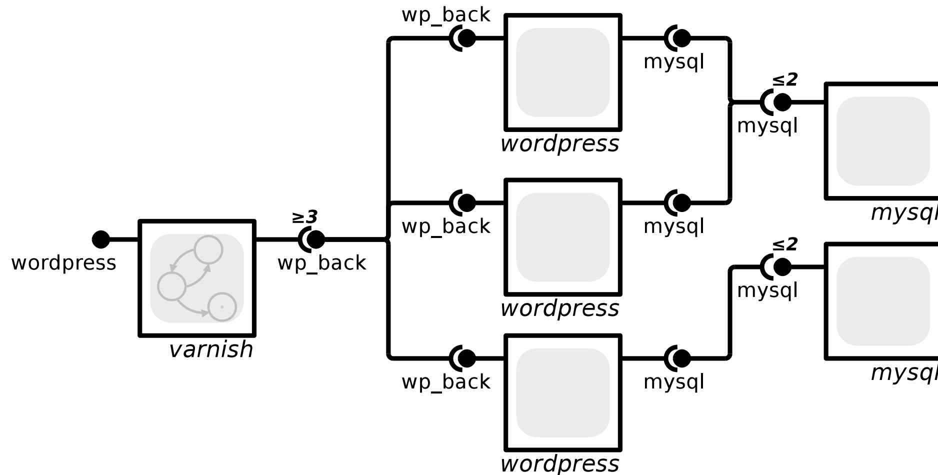


Figure 3.4: A graphical description of the model with redundancy and capacity constraints (internal sate machines are omitted for simplicity).

proxy/load balancer to the public and require to have a minimum number of *distinct* instances of Wordpress available as its back-ends.

To model this kind of situations, we allow capacity information to be added on provides and require ports of each component in Aeolus: a number n on a provide port indicates that it can fulfil no more than n requirements, while a number n on a require port means that it needs to be connected to at least n provides from n *different* components.

As an example, Figure 3.4 shows the modelling of a Wordpress hosting scenario where we want to offer high availability by putting the Varnish reverse proxy/load balancer in front of several Wordpress instances, all connected to a cluster of MySQL databases.[1] For a configuration to be correct, the model requires that Varnish is connected to at least 3 (active and distinct) Wordpress back-ends, and that each MySQL instance does not serve more than 2 clients.

As a particular case, a 0 constraint on a require means that no provide with the same name can be active at the same time; this can be effectively used to model *global conflicts* between components. For instance, we can use this feature to model the conflict between the `apache2` and `apache2.2-common` packages that had been omitted in Figure 3.2.

Use case 4 — creating and destroying components

Use cases like Wordpress hosting are commonplace in the cloud, to the point that they are often used to showcase the capabilities of state-of-the-art cloud deployment technologies. The features of the model presented up to here are already expressive enough to encode these *static* deployment scenarii, where the system architecture does not evolve over time in reaction to load changes.

[1] All Wordpress instances run within distinct Apache instances, which have been omitted for simplicity.

To model faithfully deployment runs on the cloud, where an arbitrary number of instances of virtual machine images can be allocated and deallocated on the fly, we also allow in our model creation and destruction of all kinds of components, provided they belong to some existing component type. For instance, in the configuration of Figure 3.4, to respond to an increase in traffic load one will need to spawn 2 new Wordpress instances, which in turn will require to create new MySQL instances (and bind them appropriately), as the available MySQL-s are no longer enough to handle the load increase.

3.2 The Aeolus model

We now formalize the *Aeolus component model*, implementing all the features elicited from the use cases discussed in the previous section.

3.2.1 Resources

Notation We assume given the following disjoint sets: $\mathcal{I}$ for interfaces and $\mathcal{Z}$ for components. We use $\mathbb{N}$ to denote strictly positive natural numbers, $\mathbb{N}_\infty$ for $\mathbb{N} \cup \{\infty\}$, and $\mathbb{N}_0$ for $\mathbb{N} \cup \{0\}$.

We model components as finite state automata indicating all possible component states and state transitions. When a component changes state, the sets of ports it requires from/provide to other components will also change: intuitively, the component interface with the external world varies with its state. A provide port represents the possibility of furnishing a functionality having a given interface. Similarly, a require port represent the need of a functionality with a given interface.

Definition 15 (Component type)**.** *The set* Γ *of* component types *of the Aeolus model, ranged over by* $\mathcal{T}_1, \mathcal{T}_2, \ldots$ *contains 5-ple* $\langle Q, q_0, T, P, D\rangle$ *where:*

- Q *is a finite set of states;*
- $q_0 \in Q$ *is the initial state and* $T \subseteq Q \times Q$ *is the set of* transitions*;*
- $P = \langle \mathbf{P}, \mathbf{R}\rangle$*, with* $\mathbf{P}, \mathbf{R} \subseteq \mathcal{I}$*, is a pair composed of the set of* provide *and the set of* require *ports, respectively;*
- D *is a function from* Q *to 2-ple in* $(\mathbf{P} \rightharpoonup \mathbb{N}_\infty) \times (\mathbf{R} \rightharpoonup \mathbb{N}_0)$*.*

Given a state $q \in Q$, $D(q)$ returns two partial functions $(\mathbf{P} \rightharpoonup \mathbb{N}_\infty)$ and $(\mathbf{R} \rightharpoonup \mathbb{N}_0)$ that indicate respectively the provide and require ports that q activates. The functions associate to the activate ports a numerical constraint indicating:

- for provide ports, the *maximum* number of bindings the port can satisfy,
- for require ports, the *minimum* number of required bindings to *distinct* components,
  - as a special case: if the number is 0 this indicates a conflict, meaning that there should be no other active port, in any other component, with the same name. That is, conflicts have a global nature that do not require being bound to anything to inhibit the presence of a given (active) provide.

When the numerical constraint is not explicitly indicated, we assume as default value ∞ for provide ports (i.e., they can satisfy an unlimited amount of requires) and 1 for require (i.e., one provide is enough to satisfy the requirement). We also assume that the initial state q_0 has no demands (i.e., the second function of $D(q_0)$ has an empty domain).

Example 2. Figure 3.2(a) depicts two component types: wordpress and apache2. In particular wordpress is formally defined as the 5-ple $\langle Q, q_0, T, P, D\rangle$ with:

- $Q = \{\text{uninstalled}, \text{installed}\}$,
- $q_0 = \text{uninstalled}$,
- $T = \{(\text{uninstalled} \mapsto \text{installed}), (\text{installed} \mapsto \text{uninstalled})\}$,
- $P = \langle\{\text{wordpress}\},$
  $\{\text{httpd}, \text{mysql-client}, \text{php5}, \text{php5-mysql}, \text{libphp-phpmailer}\}\rangle$,
- $D = \{(\text{uninstalled} \mapsto \langle\emptyset, \emptyset\rangle),$
  $(\text{installed} \mapsto \langle\{(\text{wordpress} \mapsto \infty)\}, f\rangle)\}$
  where f is a function that associates 1 to all require ports.

□

3.2.2 Configurations

We now define configurations that describe systems composed by component instances and bindings that interconnect them. A configuration, ranged over by $C_1, C_2, \ldots$, is given by a set of component types, a set of deployed components with a type and an actual state, and a set of bindings. Formally:

Definition 16 (Configuration)**.** *A* configuration *C is a 4-ple $\langle U, Z, S, B\rangle$ where:*

- *$U \subseteq \Gamma$ is the finite* universe *of all available component types;*
- *$Z \subseteq \mathcal{Z}$ is the set of the currently deployed* components*;*
- *S is the component* state description*, i.e., a function that associates to components in Z a pair $\langle \mathcal{T}, q\rangle$ where $\mathcal{T} \in U$ is a component type $\langle Q, q_0, T, P, D\rangle$, and $q \in Q$ is the current component state;*
- *$B \subseteq \mathcal{I} \times Z \times Z$ is the set of* bindings*, namely 3-ples composed by an interface, the component that requires that interface, and the component that provides it; we assume that the two components are distinct.*

Example 3. Figure 3.2(b) depicts a configuration with two components and one binding. Formally, it corresponds to the 4-ple $\langle U, Z, S, B\rangle$ where:

- U is a set of component types including wordpress and apache2,
- $Z = \{z_1, z_2\}$,
- $S = \{(z_1 \mapsto \langle\text{wordpress}, \text{installed}\rangle), (z_2 \mapsto \langle\text{apache2}, \text{installed}\rangle)\}$,
- $B = \langle\text{httpd}, z_1, z_2\rangle$.

□

In the following we will use a notion of configuration equivalence that relate configurations having the same instances up to renaming. This is used to abstract away from component identifiers and bindings.

Definition 17 (Configuration equivalence)**.** *Two configurations $\langle U, Z, S, B\rangle$ and $\langle U, Z', S', B'\rangle$ are equivalent, noted $\langle U, Z, S, B\rangle \equiv \langle U, Z', S', B'\rangle$, if and only if there exists a bijective function ρ from Z to Z' s.t.:*

1. *$S(z) = S'(\rho(z))$ for every $z \in Z$; and*
2. *$\langle r, z_1, z_2\rangle \in B$ if and only if $\langle r, \rho(z_1), \rho(z_2)\rangle \in B'$.*

Notation We write $C[z]$ as a lookup operation that retrieves the pair $\langle \mathcal{T}, q\rangle = S(z)$, where $C = \langle U, Z, S, B\rangle$. On such a pair we then use the postfix projection operators `.type` and `.state` to retrieve $\mathcal{T}$ and q, respectively. Similarly, given a component type $\langle Q, q_0, T, \langle \mathbf{P}, \mathbf{R}\rangle, D\rangle$, we use projections to (recursively) decompose it: `.states`, `.init`, and `.trans` return the first three elements; `.prov`, `.req` return $\mathbf{P}$ and $\mathbf{R}$; $.\mathbf{P}(q)$ and $.\mathbf{R}(q)$ return the two elements of the $D(q)$ tuple. When there is no ambiguity we take the liberty to apply the component type projections to $\langle \mathcal{T}, q\rangle$ pairs.

For example, $C[z].\mathbf{R}(q)$ stands for the partial function indicating the active require ports (and their arities) of component z in configuration C when it is in state q.

We are now ready to formalize the notion of configuration correctness:

Definition 18 (Configuration correctness)**.** *Let us consider the configuration $C = \langle U, Z, S, B\rangle$.*

We write $C \vDash_{req} (z, r, n)$ to indicate that the require port of component z, with interface r, and associated number n is satisfied. Formally, if $n = 0$ all components other than z cannot have an active provide port with interface r, namely for each $z' \in Z \setminus \{z\}$ such that $C[z'] = \langle \mathcal{T}', q'\rangle$ we have that r is not in the domain of $\mathcal{T}'.\mathbf{P}(q')$. If $n > 0$ then the port is bound to at least n active ports, i.e., there exist n distinct components $z_1, \ldots, z_n \in Z \setminus \{z\}$ such that for every $1 \leq i \leq n$ we have that $\langle r, z, z_i\rangle \in B$, $C[z_i] = \langle \mathcal{T}^i, q^i\rangle$ and r is in the domain of $\mathcal{T}^i.\mathbf{P}(q^i)$.

Similarly for provides, we write $C \vDash_{prov} (z, p, n)$ to indicate that the provide port of component z, with interface p, and associated number n is not bound to more than n active ports. Formally, there exist no m distinct components $z_1, \ldots, z_m \in Z \setminus \{z\}$, with $m > n$, such that for every $1 \leq i \leq m$ we have that $\langle p, z_i, z\rangle \in B$, $S(z_i) = \langle \mathcal{T}^i, q^i\rangle$ and p is in the domain of $\mathcal{T}^i.\mathbf{R}(q^i)$.

The configuration C is correct *if for each component $z \in Z$, given $S(z) = \langle \mathcal{T}, q\rangle$ with $\mathcal{T} = \langle Q, q_0, T, P, D\rangle$ and $D(q) = \langle \mathcal{P}, \mathcal{R}\rangle$, we have that $(p \mapsto n_p) \in \mathcal{P}$ implies $C \vDash_{prov} (z, p, n_p)$, and $(r \mapsto n_r) \in \mathcal{R}$ implies $C \vDash_{req} (z, r, n_r)$.*

Example 4. Figure 3.3 and 3.4 report examples of correct configurations. In Figure 3.3 it is easy to see that all active require ports are bound to an active provide port: this condition is enough when the numerical constraints has the default values.

In Figure 3.4 there are two kinds of non-default numerical constraints: the constraint 3 on the require port `wp_back` of the component of type `varnish` which is satisfied because

there are at least three bindings connecting it to three distinct components (we assume that the wp_back provide ports of these three components are active) and the constraint 2 on the provide port mysql of the components of type mysql which are satisfied because those ports are connected to less than two bindings. □

3.2.3 Deployment actions

We now formalize how configurations evolve from one state to another, by means of atomic actions:

Definition 19 (Actions)**.** *The set $\mathcal{A}$ contains the following actions:*

- $stateChange(z, q_1, q_2)$ *where* $z \in \mathcal{Z}$*;*
- $bind(r, z_1, z_2)$ *where* $z_1, z_2 \in \mathcal{Z}$ *and* $r \in \mathcal{I}$*;*
- $unbind(r, z_1, z_2)$ *where* $z_1, z_2 \in \mathcal{Z}$ *and* $r \in \mathcal{I}$*;*
- $new(z : \mathcal{T})$ *where* $z \in \mathcal{Z}$ *and* $\mathcal{T}$ *is a component type;*
- $del(z)$ *where* $z \in \mathcal{Z}$*.*

The execution of actions can now be formalized using a labelled transition systems on configurations, which uses actions as labels.

3.2.4 Component reconfiguration

Definition 20 (Reconfigurations)**.** *Reconfigurations are denoted by transitions $C \xrightarrow{\alpha} C'$ meaning that the execution of $\alpha \in \mathcal{A}$ on the configuration C produces a new configuration C'. The transitions from a configuration $C = \langle U, Z, S, B \rangle$ are defined as follows:*

$$\begin{array}{l} C \xrightarrow{stateChange(z,q_1,q_2)} \langle U, Z, S', B \rangle \\ \quad \text{if } C[z].\mathtt{state} = q_1 \\ \quad \text{and } (q_1, q_2) \in C[z].\mathtt{trans} \\ \quad \text{and } S'(z') = \begin{cases} \langle C[z].\mathtt{type}, q_2 \rangle & \text{if } z' = z \\ C[z'] & \text{otherwise} \end{cases} \end{array}$$

$$\begin{array}{l} C \xrightarrow{bind(r,z_1,z_2)} \langle U, Z, S, B \cup \langle r, z_1, z_2 \rangle \rangle \\ \quad \text{if } \langle r, z_1, z_2 \rangle \notin B \\ \quad \text{and } r \in C[z_1].\mathtt{req} \cap C[z_2].\mathtt{prov} \end{array}$$

$$C \xrightarrow{unbind(r,z_1,z_2)} \langle U, Z, S, B \setminus \langle r, z_1, z_2 \rangle \rangle \quad \text{if } \langle r, z_1, z_2 \rangle \in B$$

$$\begin{array}{l} C \xrightarrow{new(z:\mathcal{T})} \langle U, Z \cup \{z\}, S', B \rangle \\ \quad \text{if } z \notin Z, \mathcal{T} \in U \\ \quad \text{and } S'(z') = \begin{cases} \langle \mathcal{T}, \mathcal{T}.\mathtt{init} \rangle & \text{if } z' = z \\ C[z'] & \text{otherwise} \end{cases} \end{array}$$

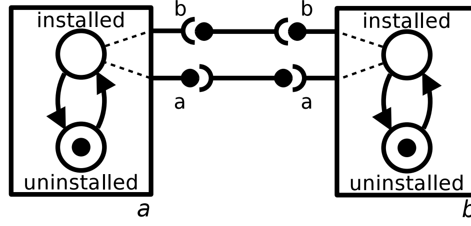


Figure 3.5: On the need of a *multiple state change*: how to install a and b?

$$
\begin{array}{l}
C \xrightarrow{del(z)} \langle U, Z \setminus \{z\}, S', B' \rangle \\
\quad \textit{if } S'(z') = \begin{cases} \bot & \textit{if } z' = z \\ C[z'] & \textit{otherwise} \end{cases} \\
\quad \textit{and } B' = \{\langle r, z_1, z_2 \rangle \in B \mid z \notin \{z_1, z_2\}\}
\end{array}
$$

Notice that in the definition of the transitions there is no requirement on the reached configuration: the correctness of these configurations will be considered at the level of deployment runs.

Also, we observe that there are configurations that cannot be reached through sequences of the actions we have introduced. In Figure 3.5, for instance, there is no way for package a and b to reach the installed state, as each package requires the other to be installed *first*. In practice, when confronted with such situations—that can be found for example in FOSS distributions in the presence of loops of *Pre-Depends* that impose an order in the installation of two depending packages—current tools either perform all the state changes atomically, or abort deployment.

We want our planners to be able to propose deployment runs containing such atomic transitions. To this end, we introduce the notion of *multiple state change*:

Definition 21 (Multiple states change)**.** *A* multiple states change *(or multi-state change)* $\mathcal{M} = \{stateChange(z^1, q_1^1, q_2^1), \cdots, stateChange(z^l, q_1^l, q_2^l)\}$ *is a set of state change actions on different components (i.e.,* $z^i \neq z^j$ *for every* $1 \leq i < j \leq l$*). We use* $\langle U, Z, S, B \rangle \xrightarrow{\mathcal{M}} \langle U, Z, S', B \rangle$ *to denote the effect of the simultaneous execution of the state changes in* $\mathcal{M}$*: formally,*

$$
\langle U, Z, S, B \rangle \xrightarrow{stateChange(z^1, q_1^1, q_2^1)} \ldots \xrightarrow{stateChange(z^l, q_1^l, q_2^l)} \langle U, Z, S', B \rangle
$$

Notice that the order of execution of the state change actions does not matter as all the actions are executed on different components.

3.2.5 Deployment runs

We can now define a *deployment run*, which is a sequence of actions that transform an initial configuration into a final correct one without violating correctness along the way. A deployment run is the output we expect from a planner, when it is asked how to reach a desired target configuration.

Definition 22 (Deployment run)**.** *A* deployment run *is a sequence* $\alpha_1 \dots \alpha_m$ *of actions and multiple state changes such that there exist* C_i *such that* $C = C_0$, $C_{j-1} \xrightarrow{\alpha_j} C_j$ *for every* $j \in \{1, \dots, m\}$*, and the following conditions hold:*

configuration correctness *for every* $i \in \{0, \dots, m\}$, C_i *is correct;*

multi state change minimality *if* α_j *is a multiple state change then there exists no proper subset* $\mathcal{M} \subset \alpha_j$*, or state change action* $\alpha \in \alpha_j$*, and correct configuration* C' *such that* $C_{j-1} \xrightarrow{\mathcal{M}} C'$*, or* $C_{j-1} \xrightarrow{\alpha} C'$*.*

Example 5. Consider the configuration reported in Figure 3.3. Starting from an empty configuration. Such configuration can be reached upon execution of the following deployment run:

1. $new(z_1 : \mathsf{wordpress})$
2. $new(z_2 : \mathsf{apache2})$
3. $stateChange(z_2, \mathsf{uninstalled}, \mathsf{installed})$
4. $bind(\mathsf{httpd}, z_1, z_2)$
5. $stateChange(z_1, \mathsf{uninstalled}, \mathsf{installed})$
6. $new(z_3 : \mathsf{mysql})$
7. $stateChange(z_3, \mathsf{uninstalled}, \mathsf{installed})$
8. $stateChange(z_3, \mathsf{installed}, \mathsf{running})$
9. $bind(\mathsf{mysql_up}, z_1, z_3)$
10. $stateChange(z_2, \mathsf{installed}, \mathsf{running})$

This sequence of actions is a deployment run because it guarantees the correctness of all the traversed configurations.

Notice that it would still be a deployment run even if step 5 were postponed. On the contrary, it would no longer be a deployment run if such action were anticipated because the requirement on the httpd port would no longer be fulfilled. It would no longer be deployment run even if such action were joined with other state changes to form a multiple state change action (like, e.g., $\{stateChange(z_1, \mathsf{uninstalled}, \mathsf{installed}), stateChange(z_2, \mathsf{installed}, \mathsf{running})\}$) because that would violate minimality. □

3.2.6 Achievability

We now have all the ingredients to define the notion of *achievability*, that is our main concern: given a universe of component types, we want to know whether it is possible to deploy at least one component of a given component type $\mathcal{T}$ in a given state q.

Definition 23 (Achievability problem)**.** *The* achievability problem *has as input a universe* U *of component types, a component type* $\mathcal{T}$*, and a target state* q*. It returns as output* ***true*** *if there exists a deployment run* $\alpha_1 \dots \alpha_m$ *such that* $\langle U, \emptyset, \emptyset, \emptyset \rangle \xrightarrow{\alpha_1} C_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_m} C_m$ *and* $C_m[z] = \langle \mathcal{T}, q \rangle$*, for some component* z *in* C_m*. Otherwise, it returns* ***false****.*

Table 3.2: Decidability and complexity of achievability in (variants of) the Aeolus component model

model	*co-domain*(.**P**())	*co-domain*(.**R**())	achievability
Aeolus	$\mathbb{N}_\infty$	$\mathbb{N}_0$	undecidable
Aeolus core	$\{\infty\}$	$\{1,0\}$	decidable, Ackermann-hard
Aeolus$^-$	$\{\infty\}$	$\{1\}$	decidable, PTIME

Example 6. Consider the achievability problem for the universe of component types wordpress, apache2, and mysql in Figure 3.3, and the target expressed by wordpress in its running state. In this case the problem returns **true** because there exists, for instance, the deployment run obtained by adding $stateChange(z_1, \texttt{installed}, \texttt{running})$ at the end of the sequence of actions in Example 5. □

Achievability is the formal property that underpins any tool whose goal is to automatically compute a way to reach a desired target configuration. It is, *mutatis mutandis*, the equivalent of the upgrade problem for FOSS distributions discussed in Chapter 2, where a package manager needed to find a way to satisfy a user request to install/remove/upgrade a (set of) package(s). The user request (there) corresponds to the target component and state (here). Note that the restriction in the achievability problem to a single component in a given state is not limiting in terms of expressivity. One can easily encode *any* given target configuration by adding a dummy provide port enabled only by the required final states of a set of components, and a dummy component with requirements on all such provides.

Similarly to what has been for packages, it is then legitimate to ask how difficult the achievability problem is. The answer is not encouraging: achievability is impossible to automatically solve in its full generality. To give a more formal and also nuanced answer, several variants of the Aeolus model have been studied. Each variant has been obtained by imposing a different restriction on the numerical constraints that are allowed as co-domains of the two $D(q)$ partial functions, making the expressivity—and conversely the difficulty of achievability—of the model vary. Table 3.2 shows the results for three variants of the Aeolus model:[2]

Aeolus (first row) is the same model of Definition 15, i.e., the full-fledged Aeolus model with its full expressivity.

Unfortunately, achievability has been proven to be undecidable for Aeolus, by reduction from the reachability problem in 2 Counter Machines (2CMs)—a well-known Turing-complete model—to Aeolus. While this of course does not mean that automatically checking achievability is impossible in *all* instances of the problem, it does make the situation much worse than what was the case for distribution packages on a single machine.

Aeolus core (second row), is a restriction of Aeolus in which provide ports always serve an unlimited amount of bindings, and require ports cannot require a minimum number of bindings strictly higher than 1. That is, in Aeolus core one can declare inter-component conflicts, but not express capacity constraints.

[2]Proofs of all these achievability results have been given in [52].

Achievability exhibits better decision properties in Aeolus core than before: it has been proven to be decidable (using the theory of Well-Structured Transition Systems [10, 70]) although Ackermann-hard (via reduction from the coverability problem in reset Petri nets, a problem of equivalent complexity [140]).

While this is a more encouraging result and puts the problem in a complexity spot similar to that of dependency solving for packages, it doesn't necessarily mean it will be easy to find a *practically efficient* solver, as it has been the case for packages thanks to SAT solving.

Aeolus^{-} (third row) is a further restriction of Aeolus core, where neither capacity constraints nor conflicts can be used: provide ports always serve an unlimited amount of bindings, and require ports are limited to require exactly 1 binding.

Achievability remains decidable (trivially from Aeolus core) and complexity further descends to PTIME, as it can be shown using an *ad hoc* forward exploration algorithm of all reachable configurations.

3.3 Bad news, good news: a compromise

In a sense, we have only gathered bad news with the decidability and complexity of achievability in Aeolus. When taking into account all expressivity requirements elicited from real needs we end up with a model in which achievability is undecidable. We can get back to a more amenable model by giving up capacity constraints, but the complexity of the resulting problem is still daunting. To scale further down to a fully manageable problem we have to also give up on the ability to express conflicts, which results in a model with very poor expressivity.

These limiting results also offer a key for reviewing the state of the art of industrial tools meant to instrument automatic deployment of software components in cloud and networked settings. Unsurprisingly, industrial tools tend to be close to the expressivity of Aeolus^{-}. They generally support scaling up and down the number of components but offer no way of enforcing capacity constraints; it is up to the user to notice, possibly via companion monitoring tools, the need of increasing/decreasing capacity. Also they tend to avoid inter-machine conflicts, generally by relegating them to host boundaries, as supported by conflict among plain old distribution packages.

Can we do any better in terms of tooling for automated deployment of components in the cloud, with a better expressivity than Aeolus^{-}? The good news, elaborated in the next sections, is that indeed we can, but only at the price of giving up on completeness. The main idea is to decouple the *provisioning* (i.e., creation and or destruction) of the components that need to be deployed in the target configuration to satisfy user request, from the planning of the sequence of *state changes* that are needed to bring the current configuration to the target one. Completeness is no longer guaranteed in this approach because it is theoretically possible that the provisioning step will propose a set of components that cannot be brought to the desired state with a correct deployment run that is only allowed to change component states, e.g., due to conflicts or capacity constraints. In those cases, though, experienced system administrators and devops would in generally be able to fix the problem "by hand", and would have still benefited from automation in the first, arguably more difficult, step.

This approach has been realized by tools that have been created as part of the Aeolus ANR project, all available [12] and released under FOSS licenses.

Zephyrus tackles the problem of computing a valid system configuration (according to the semantics of the Aeolus model), starting from an existing configuration, a universe of available component types, and a formal specification that captures user desiderata for the target system. Furthermore, Zephyrus also takes into account limited machine resources, such as CPU, memory, bandwidth, etc. The computation is done via translation to a set of integer constraints, plus a dedicated algorithm to compute bindings, and the approach makes it is possible to add an objective function that can be minimized or maximized to optimize the resulting configuration.

Zephyrus is briefly discussed in the remainder of this chapter. Full details about Zephyrus are available in [49, 50].

Metis takes care of the complementary problem of quickly computing a plan that migrates a valid Aeolus configuration into a different one, possibly synthesized by Zephyrus.

Metis has been designed and implemented by Lascu, Mauro, and Zavattaro and is further described in [104, 105].

Armonic closes the gap between configuration and deployment plans produced by the above tools, and the technological reality of virtual machines and cloud platforms. Armonic takes a full system configuration, possibly produced by Zephyrus, and deploys it by provisioning the required virtual machines on a cloud computing platform, installing the needed packages, configuring the various services at the configuration-file level, and starting them in the right order.

Armonic is by Antoine Eiche and the Mandriva company an available for download at `https://github.com/armonic/armonic`. Further details about Armonic are available in [50].

3.4 Automated cloud deployment with Aeolus

Zephyrus uses a stateless version of the Aeolus model [49], extended to take into account locations (i.e., real or virtual machines where packages will be installed), package repositories (possibly from different FOSS distributions), packages, and resources. Packages and repositories are encoded following the approach of Chapter 2.

The specifications accepted by Zephyrus are given in a rigorous syntax whose semantics defines when a configuration satisfies a specification. Based on this formalization, Zephyrus has been proven correct and complete: it will always find a configuration that is optimal with respect to the chosen criterion if one exists. Furthermore, the generated configuration is guaranteed to provide the expected functionalities, and satisfies the constraints defined by the replication policies, as well as the dependencies and conflicts between services.

Zephyrus takes several inputs:

1. a description of all the existing components and their constraints, which come in various formats due to their different origins (e.g. package database, architectural choices, machine physical resources, etc.); this is called a *universe*.

2. a description of the *current system configuration* (existing machines, which services are currently deployed where, etc.)
3. a high level *specification* of the desired target system. As part of the specification, architects can include objective functions that they would like to optimize for, such as the desire of minimizing the number of virtual machines that will be used for the deployment (and hence the system cost).

Let's see all this in action in a realistic cloud deployment use case.

The task we want to perform is deploying Wordpress on a private OpenStack [127] cloud. Wordpress is written in PHP and as such is executed within Web server software like Apache or nginx. Additionally, Wordpress needs a connection to a MySQL instance, in order to store user data. Simple Wordpress deployments can therefore be obtained on a single machine where both Wordpress and MySQL get installed.

"Serious" Wordpress deployments, however—that sustain high load and are fault tolerant—are more complex and rely on some form of load balancing. One possibility is to balance load at the DNS level using servers like Bind: multiple DNS requests to resolve the website name will result in different IPs from a given pool of machines, on each of which a separate Wordpress instance is running. Alternatively one can use as website entry point an HTTP reverse proxy capable of load balancing (and caching, for added benefit) such as Varnish. Either way, Wordpress instances will need to be configured to contact the same MySQL database, to avoid delivering inconsistent results to users. Also, having redundancy and balancing at the front-end level, one usually expects to have them also at the DBMS level. One way to achieve that is to use a MySQL *cluster*, and configure the Wordpress instances with multiple entry points to it.

Several design constraints should be taken into account when designing such a system. Some constraints come from package providers and cannot be easily changed. For instance, Wordpress, Varnish, etc. usually come from software distribution packages and have their own dependencies and conflicts which must be respected on each machine when installing the software. On the other hand, "house" requirements are defined by system architects to capture some ad-hoc policy. For this use case, we assume given the following requirements:

- at least 3 replicas of Wordpress behind Varnish or, alternatively, at least 7 replicas with DNS-based load balancing (since DNS-based load balancing is not capable of caching, the expected load on individual Wordpress instances is higher);
- at least 2 different entry points to the MySQL cluster;
- each MySQL instance shouldn't serve the needs of more than 3 Wordpress instances;
- no more than 1 DNS server deployed in the administrative domain;
- different Wordpress (and MySQL) instances are deployed at different locations.[3]

Similar constraints might exist on machine resources, e.g., we expect Varnish to consume 2GB of RAM and we don't want to deploy it to a smaller machine, especially if in combination with other RAM-consuming services. Note that "house" requirements are not

[3]It is technically possible to co-locate multiple, say, MySQL instances on the same machine, but it would be pointless to do so when we are seeking fault tolerance and load balancing.

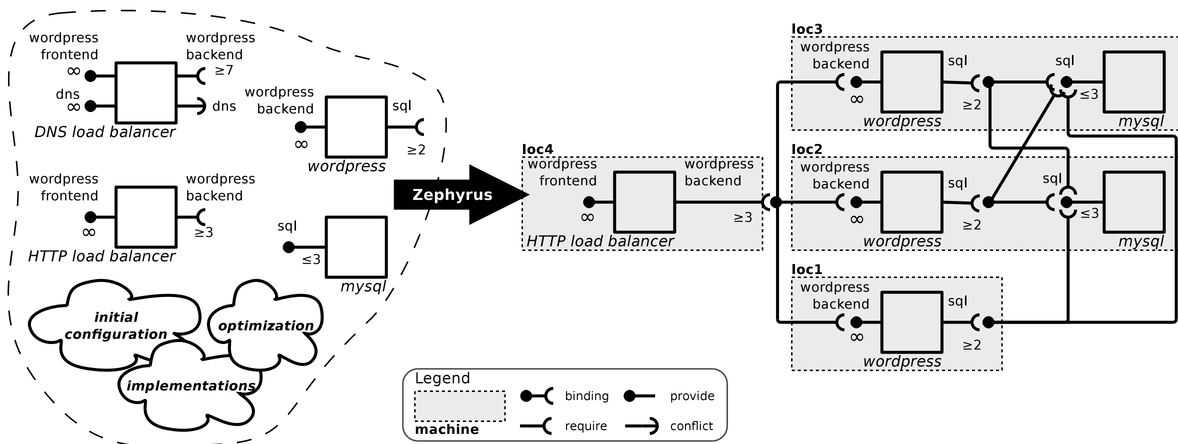


Figure 3.6: Zephyrus used to design a scalable, fault-tolerant Wordpress deployment

intrinsically related to the software components we are using, but are rather an encoding of explicit architectural choices.

3.4.1 Architecture synthesis

Figure 3.6 shows the application of the Aeolus toolchain to the design of the discussed Wordpress deployment.

On the left of the black arrow is a schematic representation of Zephyrus input, on the right its output. Available services are depicted in the figure using a graphical syntax of the Aeolus model, each one with its own requirements, conflicts, and (house) replication policy. In the figure, the HTTP load balancer requires 3 Wordpress replicas and the DNS load balancer is incompatible with any other DNS service. Note that our ports result in a very flexible notion of dependency, with *choice*: any requirement can be satisfied by *any* component providing the right port. For instance, if we require the port `wordpress frontend`, we allow Zephyrus to choose which of `DNS load balancer` or `HTTP load balancer` is the best to use. To our knowledge, Zephyrus is the only tool to manage such flexibility in dependencies.

Services and implementations

Zephyrus takes as input a description of the available service types, and an *implementation* relation that maps each *service* to the set of packages implementing it.[4] These two parts of the universe are given as input to Zephyrus as a JSON file that for our example looks like this:

```
{ "component_types": [
  { "name"    : "DNS-load-balancer",
    "provide" : [["@wordpress-frontend"], ["@dns"]],
    "require" : [["@wordpress-backend", 7]],
```

[4] In the example we have kept things simple, but Zephyrus is capable of handling complex situations where the same service can be implemented by different packages on different machines, according to the locally installed OS.

```
    "conflict": ["@dns"],
    "consume" : [["ram", 128]] },
  { "name"    : "HTTP-load-balancer",
    "provide" : [["@wordpress-frontend"]],
    "require" : [["@wordpress-backend", 3]],
    "consume" : [["ram", 2048]] },
  { "name"    : "Wordpress",
    "provide" : [["@wordpress-backend"]],
    "require" : [["@sql", 2]],
    "consume" : [["ram", 512]] },
  { "name"    : "MySQL",
    "provide" : [["@sql", 3]],
    "consume" : [["ram", 512]] } ],
  "implementation": [
    [ "DNS-load-balancer", ["bind9"] ],
    [ "HTTP-load-balancer", ["varnish"] ],
    [ "Wordpress", ["wordpress"] ],
    [ "MySQL", ["mysql-server"] ] ] }
```

The `component_types` section describes the available component types with their ports, as well as their non functional requirements like memory or bandwidth. Port names are distinguished from components or packages by a simple syntactic convention: ports start with `@`. The `implementation` section maps services to the software packages that should be installed to realize them on actual machines.

Package repositories

Zephyrus is fully aware of available package repositories, with their dependencies and conflicts, and uses such information to ensure that package-level conflicts and dependencies are respected on all machines. It is possible to associate different package repositories to different locations, allowing to handle deployment of heterogeneous systems. As the size of a repository might be huge, Zephyrus uses `coinst` [55] to abstract packages into a set of much smaller equivalence classes but yet sufficient to capture all package incompatibilities.

Available (virtual) machines

Another essential part of Zephyrus input is the description of the initial configuration, i.e., the set of available machines with information on their resources: memory, package repository, existing services and packages. In our example, we start with an initial configuration consisting of 6 bare locations with 2GB of RAM. Such configuration is fed to Zephyrus in JSON format, e.g.:

```
{ "locations" : [
    { "name" : "loc1",
      "repository" : "debian-squeeze",
      "provide_resources" : [["ram", 2048]] },
    { "name" : "loc2",
      "repository" : "debian-squeeze",
      "provide_resources" : [["ram", 2048]] },
    [...] }
```

Table 3.3: Zephyrus specification abstract syntax

S	::=	`true` $\mid$ $e\ op\ e$	Specification
	$\mid$	S `and` S $\mid$ S `or` S	
	$\mid$	S `=>` S $\mid$ `not` S	
e	::=	n $\mid$ $\#\ell$ $\mid$ $n \times e$	Expression
	$\mid$	$e + e$ $\mid$ $e - e$	
ℓ	::=	k $\mid$ t $\mid$ p	Elements
	$\mid$	$(J_\phi)\{J_r : S_l\}$	
S_l	::=	`true` $\mid$ $e_l\ op\ e_l$	Local Specification
	$\mid$	S_l `and` S_l $\mid$ S_l `or` S_l	
	$\mid$	S_l `=>` S_l $\mid$ `not` S_l	
e_l	::=	n $\mid$ $\#\ell_l$ $\mid$ $n \times e_l$	Local Expression
	$\mid$	$e_l + e_l$ $\mid$ $e_l - e_l$	
ℓ_l	::=	k $\mid$ t $\mid$ p	Local Elements
J_ϕ	::=	_ $\mid$ $o\ op\ n; J_\phi$	Resource Constraint
J_r	::=	_ $\mid$ r $\mid$ $r \vee J_r$	Repository Constraint
op	::=	$\leq$ $\mid$ $=$ $\mid$ $\geq$	Operators

Target system specification

Zephyrus accepts a specification of the desired target system. Specifications are defined according the abstract syntax presented in Table 3.3.

A specification S is a set of basic constraints $e\ op\ e$, combined using the usual logical connectors. These basic constraints specify how many elements (packages, component types, etc) should be in the generated configuration, using terms of the form $\#\ell$ that correspond to the number of instances of element ℓ in the system. For instance, one might state that we want at least 3 instances of the component type `apache`: "`#apache` ≥ `3`", where `#apache` represents the number of `apache` instances in the configuration.

Moreover, it is possible to express constraints on locations. Locations can be specified in our syntax with the term $(J_\phi)\{J_r : S_l\}$ where J_ϕ is the constraint on the resource available on that machine; J_r is the set of repositories that can be installed on that machine ('_' standing for any repository); and S_l is a constraint specifying the contents of the machine (basically, S_l is S without locations). For instance, we can specify that no location with less than 2GB of RAM and `redhat` installed should have a `MySQL` running: "`#(mem < 2G){redhat: #MySQL` ≥ `1} = 0`".

For our running example we need exactly one Wordpress frontend (i.e., exactly one service offering a `wordpress-frontend` port), and that no machine is deployed with more than one instance of either MySQL/Wordpress services on it.

```
(#@wordpress-frontend = 1)
and #(_){_ : #MySQL > 1} = 0
and #(_){_ : #Wordpress > 1} = 0
```

Note that no constraint is imposed on the co-location of *different* services on the same machine.

Optimization criteria

In Zephyrus, one may request a solution that is optimal with respect to a specific objective function. Currently, Zephyrus supports two built-in optimization criteria, namely `compact` and `conservative`, which respectively minimize the number of components and locations used (a cost efficient criterion), or their *difference* with respect to the initial configuration (similar to the *paranoid* use case discussed for packages in Chapter 2).

Running Zephyrus

We are now ready to ask Zephyrus to compute the final configuration:

```
$ zephyrus -u univ.json -opt compact \
   -ic conf.json -spec sp.spec \
   -repo debian-squeeze ds.coinst
```

In addition to the obvious parameters (universe, optimization function, configuration, specification), we pass an extra one: the `-repo` option tells Zephyrus that all the information about the packages contained in the repository named `debian-squeeze` is available in the file `ds.coinst`.

The actual output of Zephyrus contains a complete description of the system to be deployed; it is too long to be listed here in full, so we only highlight some excerpts of it. The format is the same as for configurations, and starts with the description of the locations:

```
{ "locations": [
    { "name": "loc1",
      "provide_resources": [ [ "ram", 2048 ] ],
      "repository": "debian-squeeze",
      "packages_installed": [ "wordpress" ] },
    { "name": "loc2",
      "provide_resources": [ [ "ram", 2048 ] ],
      "repository": "debian-squeeze",
      "packages_installed": ["mysql-server",
        "wordpress" ] }                         [...]
```

We see that each location is associated to a list of packages that should be installed there. Only the root packages are listed, and Zephyrus has already checked that they can be co-installed, satisfying dependencies and conflicts.

The second part of the output is the list of service instances present in the system, mapped to their locations:

```
  "components": [
    { "name": "Wordpress-1",
      "type": "Wordpress",
      "location": "loc1" },
    { "name": "Wordpress-2",
      "type": "Wordpress",
      "location": "loc2" },
    { "name": "MySQL-1", "type": "MySQL",
      "location": "loc2" },                 [...]
```

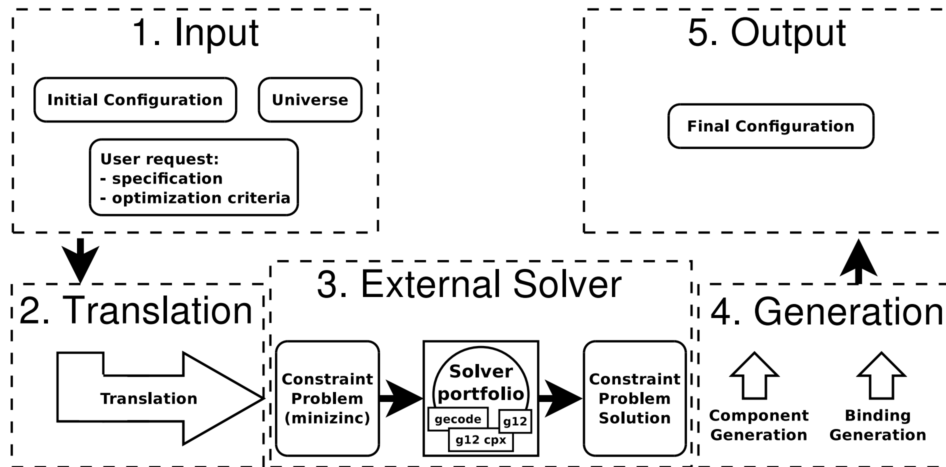


Figure 3.7: Internal architecture of Zephyrus

Finally, the third part of the output lists the bindings that connect (ports of) service instances together:

```
"bindings": [
  { "port": "@wordpress-backend",
    "requirer": "HTTP-load-balancer-1",
    "provider": "Wordpress-1" },
  { "port": "@wordpress-backend",
    "requirer": "HTTP-load-balancer-1",
    "provider": "Wordpress-2" },
  { "port": "@sql",
    "requirer": "Wordpress-1",
    "provider": "MySQL-1" }                [...]
```

The configuration corresponding to Zephyrus output is depicted on the right of Figure 3.6, where shaded boxes denote locations; we omit installed packages for the sake of readability. All choices there—load balancer, mapping between services and machines, bindings, etc.—have been made by Zephyrus. Note how services have been co-located where possible, minimizing the number of used machines: only 4 out of the 6 available machines have been used: the proposed solution is optimal with respect to the desired metric.

The obtained configuration could then be deployed automatically, including virtual machine provisioning and package installation, using Armonic.

3.5 Aeolus toolchain internals

3.5.1 Minimizing input

Figure 3.7 presents a simple schema of the architecture of Zephyrus, which is basically structured into five blocks. The input phase of Zephyrus collects all the data provided by the user.

For each location, we take into account not only the available services, but also all the possible ways to deploy them (i.e., packages that must be installed to realize them, together with their dependencies): this can amount to handle tens of thousands of packages for each location, and a naive approach would simply be unfeasible. This is likely one of the fundamental reasons why competing tools do not represent package relationships explicitly or completely, with the consequence of potentially producing configurations which are not deployable due to package incompatibilities unknown to the tool.

To render the problem tractable, Zephyrus performs several simplification passes on the input data that greatly reduce its size: the universe is trimmed by removing all services that are not in the transitive closure of the services present in the initial configuration or the request; package repositories are pruned by keeping only packages that implement services which were not removed by the previous simplification phase; lower and upper bounds on the needed resources and components are computed, and only the minimum estimated number of available locations is kept. All these operations are safe, as we can prove that they do not exclude any correct solution.

A second important simplification is achieved by using a slightly modified version of the `coinst` tool [55], which reduces by several orders of magnitude the data present in software package repositories, like those offered by the Debian or RedHat distributions, while retaining all the *coinstallability* information needed to determine if a set of packages can or cannot installed together.

3.5.2 Constraint generation

The second phase of Zephyrus translates the (trimmed) input into a set of constraints over non-negative integers. These constraints use different variables for the number of instances to create on each of the locations for each of the types in the universe, and also variables representing the packages that must be installed on each location. The constraints impose that the instances respect the definition of their type in the universe, the way how instances are implemented by packages, the (compacted) dependencies and conflicts between packages, and the problem specification.

The most interesting of these constraints ensure that it is possible to create the bindings between all instances according to the capacity constraints. These constraints, missing from competing approaches [68, 67, 71], are necessary due to the flexible dependencies that Aeolus allows on ports. Constraints are constructed using auxiliary variables $B(p, t_r, t_p)$ for the number of bindings on port p between requesting instances of type t_r and providing instances of type t_p.

On our Wordpress example, these particular constraints for the bindings on port `sql` look like this:

$$B(\texttt{sql},\texttt{wp},\texttt{mysql}) \leq \#\texttt{mysql} * 3 \quad (3.1)$$
$$B(\texttt{sql},\texttt{wp},\texttt{mysql}) \geq \#\texttt{wp} * 2 \quad (3.2)$$
$$B(\texttt{sql},\texttt{wp},\texttt{mysql}) \leq \#\texttt{wp} * \#\texttt{mysql} \quad (3.3)$$

Here, (3.1) expresses that the number of bindings on port `sql` between instances of the two types is at most the number of instances of the providing type `mysql` times 3 (since

any component of that type can bind to at most 3 instances), (3.2) that the number of bindings on port `sql` is at least the number of instances of the requesting type `wp` times 2 (the number of binding each component of type `wp` requires), and finally (3.3) states that the number of bindings is at most the number of pairs of instances of type `wp` with instances of type `sql`. This last restriction expresses that no two bindings may exist between the same pair of instances.

The ability to capture as simple integer constraints the existence of a complete architecture corresponding to a specification is the cornerstone of Zephyrus approach. It allows to deal with the many facets of system design as a whole, and thus ensures the completeness of architecture synthesis (in the stateless model) as well as the optimality of the generated configuration. In particular, if the output of Zephyrus indicates that several services are to be installed on the same machine, we know that no conflict between the packages that realize them will arise on actual machines.

3.5.3 External solvers

The generated constraints, as well as the optimization function, are expressed using the `MiniZinc` constraint modelling language [118, 115]. This allows to employ any of the many existing constraint solvers that support `MiniZinc`.

Zephyrus can currently use GeCode [141] (an efficient FOSS solver) or several of the solvers provided by the G12 suite [150]. The tool exploits this flexibility by implementing a *solver portfolio* approach [14, 15] that reduces execution time by running several solvers in parallel and stops as soon as one of the solvers finds a (optimal) solution.

3.5.4 Configuration generation

When the external solver finds a solution for the generated constraints, the next part of Zephyrus transforms that solution, consisting in a simple mapping from variables to integers, to an actual Aeolus configuration. The two main challenges for this step are: i) reuse as many existing parts of the initial configuration as possible in order to minimize the impact on the existing system; and ii) correctly generate bindings between the instances taking into account that any two instances can be bound on a given port at most once. The two challenges are addressed by an *ad hoc* algorithm presented in detail in [49].

Once the configuration has been generated, it can be written to a file in two different formats: either (a) the same *JSON* format used for the input configuration, which precisely describes all the configuration features and is used by Armonic as input; or (b) the *Graphviz* format that encodes the configuration into a graph that can be viewed using the `dot` program to visualize the synthesized architecture.

If no configuration can satisfy the input constraints, Zephyrus will exit with an error message and produce no output files.

3.5.5 Synthesis soundness and completeness

An important property of Zephyrus is that all its parts have been formalized. In particular, the translation into constraints and the generation of the configuration have been

precisely described and proven correct in [49], where the following results have been shown for Zephyrus:

Theorem 2 (Soundness)**.** *The configuration generated by Zephyrus is correct with respect to the input universe and specification.*

Theorem 3 (Completeness)**.** *If there exists a configuration that validates the input universe and specification, then Zephyrus will successfully generate a correct configuration.*

Note that completeness here is with respect to the stateless variant of the Aeolus model that is used by Zephyrus.)

Theorem 4 (Optimality)**.** *The configuration generated by Zephyrus is optimal with respect to the input optimization function.*

3.6 Validation

Zephyrus have been experimented with in both artificial benchmarks and real industrial settings.

3.6.1 Synthesis efficiency

Several architecture synthesis benchmarks have been conducted—as part of the Ph.D. thesis of Jakub Zwolakowski [171] that we have supervised—on both realistic and extreme use cases.

To that end the Wordpress use case has been parameterized to scale it up and demonstrate how Zephyrus handles problems which require more and more components and locations: (i) the first parameter is the minimum replication constraint on the `wordpress backend` port (required by the load balancer); (ii) the second parameter is the minimum replication constraint on the `sql` port (required by Wordpress). Both parameters have been made to vary from 1 to 16.

The resources associated to available locations have been inspired by Amazon's EC2 VM instance types: `m1.small`, `m1.medium`, `m1.large` and `m1.xlarge`. Zephyrus was provided with a finite, but large enough number of available machines (250 for each instance type). A single package repository was associated to each machine, and a single package implementing all the available component types.

A portfolio of solvers has been used, consisting of 3 solvers: GeCode [141], the standard finite domain constraint solver from the G12 suite [150], and the G12/CPX (Constraint Programming with eXplanations) solver from the G12 suite. As these solvers are optimized for different goals, each of them works better in some situations and worse in others.

Execution times for architecture synthesis, shown in Figure 3.8, have been obtained as the average of 5 Zephyrus runs on a commodity desktop machine at the time. Results show that the vast majority of cases are solved very quickly in less than one minute. Only the larger ones can take more than 20 minutes, e.g., the `(14,16)` case, which is the highest peak in the chart. To put this worst-case into perspective, note that the largest

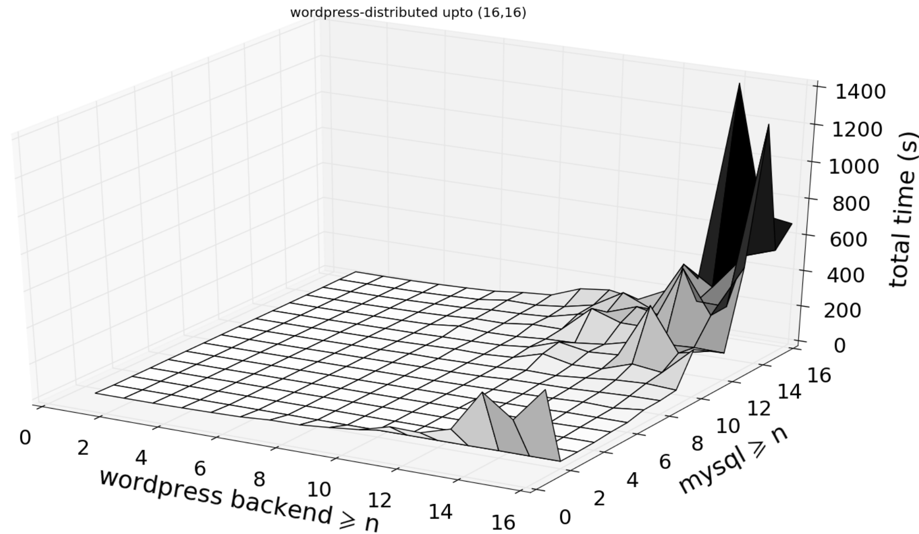


Figure 3.8: Wordpress synthesis benchmarks, for increasing values of the replication constraints on Wordpress backends and MySQL

use case ((16,16)) consists of 103 components, interconnected by 272 bindings, and distributed over 86 machines. This surpasses by a significant margin the size of most professional Wordpress deployments.

3.6.2 Industry adoption

In addition to use in production as part of cloud provisioning products by the Mandriva company (the authors of Armonic), Zephyrus has also been deployed in a large industrial use case at Kyriba Corporation,[5] a large software editor providing Software-as-a-Service treasury management solutions.

Kyriba solution is a complex software platform composed of more than 150 components deployed on multi-tier architectures, with many different versions running at the same time. Maintaining the consistency of the system as a whole is a major undertaking. To address this challenge, Kyriba has invested in completely automating the build, integration, and deployment processes.

Successful completion of a local qualification process (on developers machines), is required in order to be able to commit code changes to the source version control system. After commit, a remote qualification process is used by first rerunning on a Continuous Integration (CI) [62] pipeline the local qualification process, and then by performing automatic deployment and deep functional testing on a private cloud of all the latest component versions. The testing process is very time consuming: while local tests take less than 4 minutes to complete, global ones might take 4–8 hours.

Test deployment was historically done using custom tools involving a manual setup, and component/protocol incompatibilities were only detected at runtime. Short feedback

[5] http://www.kyriba.com/

loops discipline helps developers with error diagnostic related to small code changes [88], so Kyriba has been looking for a tool that could anticipate error detection. Zephyrus turned out to be a perfect fit for this need, as a deployment validation tool for both the local and remote qualification processes. Zephyrus is now used in distributed component consistency validation and deployment configuration scenarios. Zephyrus helps to get feedback before launching local deployments tests and has led to a significant reduction of the number of failures occurring during automated deployment in comparison to the previous, more manual, test setup.

Here is how Kyriba tech lead summed up their experience with Zephyrus:

> *"Instead of managing deployment scenarios manually using spreadsheets and flat documents, with* ad hoc *semantics leading to complex, time-consuming and error-prone deployments, Zephyrus brings precise semantics and simplifies the automation of software qualification processes. Zephyrus provides static validation before actually performing expensive and very long dynamic validation at runtime. As most "compiler-like" tools, Zephyrus improves engineering quality and reduces building cost with less failures at deployment, integration test stage and platform upgrade." [50]*

Chapter 4

Delving into the source code of large FOSS collections

This chapter is based on [31, 30].

The kinds of modeling and analysis of FOSS components discussed in Chapters 2 and 3 stop at the abstraction level of packages and rely on the fact that their declared contextual requirements (dependencies, conflicts, etc.) match the reality of the actual software they stand for. At such an abstraction level it is possible to spot "global" inconsistencies and plan deployments, but not actually detect "local" mismatches between declared package relationships and the underlying software. To attack such a problem we need to delve into the *content* of FOSS packages, i.e., their source code.

Several other problems that relate to topics briefly touched by the Mancoosi and Aeolus projects can also be solved only at the source code level. A notable example is early detection of upgrade runtime failures [54], as might be induced by buggy *maintainer scripts* [45, 92].

The next natural question is then how to best analyze—or "mine", in the jargon of the Mining Software Engineering (MSR) community [82, 96]—the source code of large, curated software collections like FOSS distributions. Ideally, we would like to have a solution that does not only serve the immediate need of *a* specific study, but rather find a *generic* solution that can then be leveraged by scholars around the world to attack different problems over time.

Aside from this generality requirement, we observe that the study of software collections poses specific challenges for scholars, due to their 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.

In this chapter we present our work on Debsources, that has been a first contribution to solve these problems. Debsources aims at easing the realization of FOSS studies through the lenses of the Debian distribution, which is already recognized as a popular subject of empirical software engineering studies [43]. Specific attention has been devoted to support software evolution studies [151, 26], both long-term studies—looking back as far as possible—and studies of present, day-by-day evolution patterns of software currently shipped by Debian. But the resulting platform, called *Debsources*, is more

general than software evolution and eases all kind of large-scale analyses on the entire source code shipped by Debian—presently and in the past—minimizing the efforts required by researchers.

From the point of view of researchers, Debsources is in fact two things at once. First and foremost, it is an extensible software platform (described in Section 4.1) that allows to gather, search, and publish on the Web the source code of Debian as well as measures about it. The most notable instance of Debsources is available at `http://sources.debian.net` (or `sources.d.n` for short). It contains the source code, and metadata about it, of all historical Debian releases. `sources.d.n` is also integrated with the official Debian infrastructure and, as such, receives live updates of new packages as they hit the Debian archive. Debsources is FOSS[1] released under the AGPL3 license; it can be deployed elsewhere, possibly targeting other Debian-based distributions, to serve similar needs.

Second, Debsources is a curated, open dataset (discussed in Section 4.2), obtained as a byproduct of maintaining `sources.d.n`, that contains source code and related metadata spanning two decades of FOSS history. The Debsources Dataset spans more than 3 billion lines of source code as well as metadata about them such as: size metrics (lines of code, disk usage), developer-defined symbols (ctags), file-level checksums (SHA1, SHA256, TLSH), file media types (MIME), release information (which version of which package containing which source code files has been released when), and license information (GPL, BSD, etc).

A large-scale evolution case study is presented in Section 4.3 to showcase how the Debsources Dataset can be used to easily and efficiently instrument large-scale analyses of Debian from various angles (size, granularity, licensing, etc.), getting a grasp of major FOSS trends of the past two decades.

4.1 The Debsources platform

4.1.1 Architecture

The life-cycles of Debian packages and releases are depicted in Figure 4.1. Abstracting over those life-cycles, so that researchers using the platform don't have to worry too much about those details, was one of the key design goal of Debsources.

Aside from the software integration steps that distribution notably performs with respect to upstream software authors (leftmost part of Figure 4.1) and final users (rightmost part), the most relevant parts of Debian life-cycles are release management and archival (inner part of the figure).

When a new package version is ready, the corresponding package maintainer uploads both source and binary packages to the development release (or "suite") called *unstable* (a.k.a. *sid*). Note that Debsources is only concerned with source packages; therefore, throughout this chapter and unless otherwise specified, we use "package" to mean "*source* package". Since Debian supports many hardware architectures, a network of

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

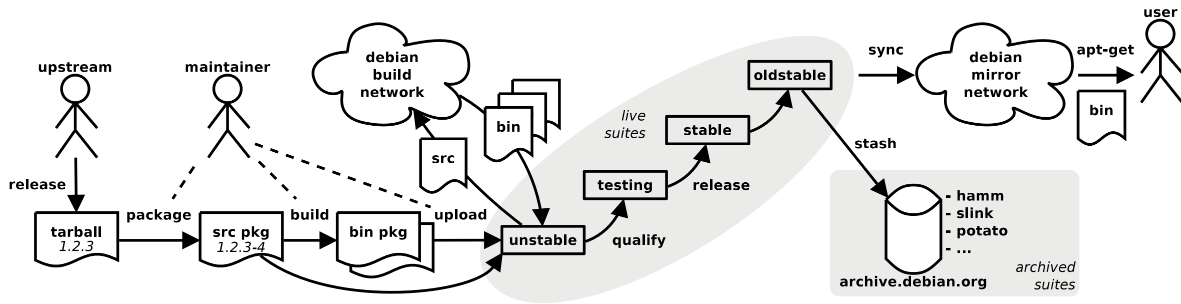


Figure 4.1: Life-cycle of Debian packages and releases

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* [164], 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`, or `archived.d.o` for short—which is separately mirrored and contains all historical releases.

For reference, Table 4.1 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 now 590 days (respectively 768 over the past 15 years, since *woody*) with a standard deviation of 269 days (respectively 124 days).

The architecture of Debsources and its data flow are depicted in Figure 4.2. On the back end, Debsources inputs are the mirror network (for live suites) and `archived.d.o` (for archived ones). Live suites can be mirrored running periodically (e.g., via `cron`) the dedicated `debmirror` tool,[2] 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 the long time period we have to take into account.

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.[3]

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

[2] `http://packages.debian.org/sid/debmirror`
[3] push notifications are enabled on the `sources.d.n` Debsources instance

Table 4.1: Debian release information; * denotes, here and in the remainder, unreleased suites.

version number	release name	current alias	release date	cycle duration (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		06/02/2011	721	yes
7	wheezy	oldoldstable	04/05/2013	818	no
8	jessie	oldstable	26/05/2015	752	no
9	stretch	stable	17/06/2017	753	no
10	buster*	testing	*tbd*	*tbd*	no
n/a	sid*	unstable	*n/a*	*n/a*	no

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 PostgreSQL [149] database, which contains information about package metadata, suites, and individual source files.

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 or use general purpose plugin tables, such as `metrics` (see Figure 4.3 later in this chapter for the database schema).

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. Plugins are available to compute popular source code metrics: disk usage, physical source lines of code (SLOC) using `sloccount` [167], user-defined "symbols" (functions, classes, types, etc.) using Exuberant Ctags,[4] different types of checksums of all source files, etc. Note that simpler metrics like the number of source files do not need specific plugins, because Debsources natively tracks individual files.

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

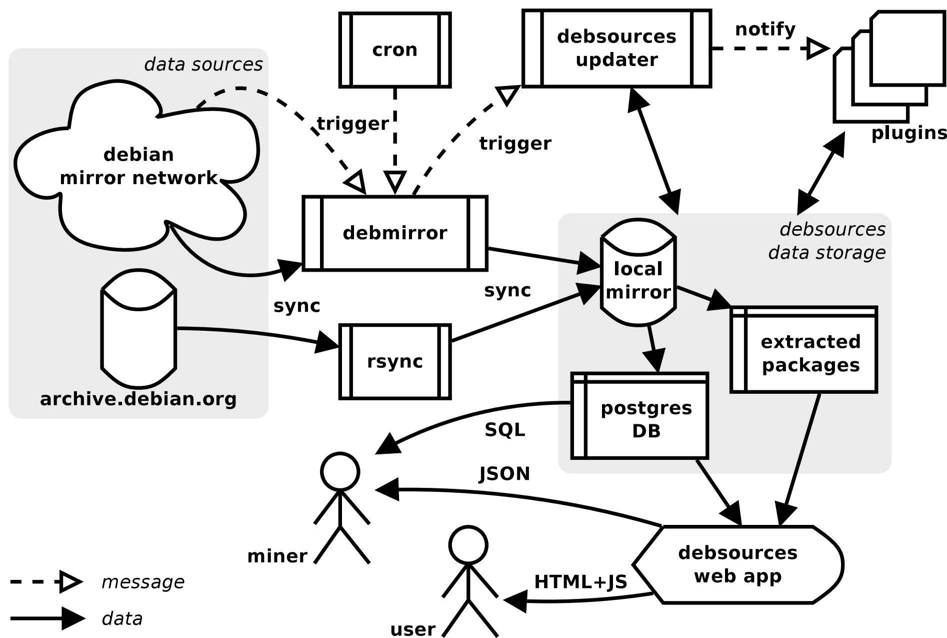


Figure 4.2: Debsources architecture

Plugins are quick and easy to write: 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 the plugins mentioned above are already 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, metadata searches, and regular expression searches on the code via Debian Code Search [146]. The same features are exposed to developers via a JSON-based Web API. Additionally, scholars interested in aggregate queries can directly access the low-level Debsources DB using SQL.

4.1.2 Adoption

The Debsources instance running at `sources.d.n` has quickly been adopted by the Debian community as the reference Web platform where to search and browse Debian source code, in particular to support developers in their day-by-day activities.

Web hit ratio for `sources.d.n` has been steadily growing and currently averages at around 500 000 pages requests per month. `sources.d.n` is also regularly used on developer communication channels (mailing lists, IRC, etc.) to point other developers to specific lines of code, as examples of how to achieve something (code reuse), or as a way to pinpoint where bugs that need to be fixed can be found.

The success of these use patterns was, in retrospect, to be expected. Before `sources.d.n` the only alternative to point other developers to specific lines of code in Debian was to communicate package name, version, file name, and line number; expecting then the receiving end of the communication to go fetch the package, extract it (which

might result in GB of disk occupation for large packages), and manually resolve the given identifiers.

Another qualitative measure of the success of `sources.d.n` is Debian is the extent to which it has been integrated, by developers other than the Debsources authors, with other services in the project infrastructure. A few examples are the already mentioned integration with Debian Code Search [146], that searches the code of the development suite of Debian and relies on `sources.d.n` to show search results in context; and the integration with the Debian package tracker that added a "browse source code" functionality using `sources.d.n` as backend.[5]

As a FOSS project, Debsources is also quite healthy. Since its public announcement the project has collected 13 contributors and has been the subject of 5 internships, one of which academic (supported by Inria) and 4 of which supported by FOSS outreach/mentoring programs such as Google Summer of Code and Outreachy.

Thanks to those internships Debsources has also gained new interesting features that are starting to be exploited by industry consortiums active in the FOSS space. Most notably, Debsources can now detect and extract from packages software licensing information (see Section 4.3.4 later on in this chapter for details), associate them to individual source code files, and export the results in standardized formats. Industry members of the SPDX (Software Package Data Exchange [148]) consortium are starting to use `sources.d.n`-originated licensing information as part of their software qualification tooling and processes.

4.2 The Debsources Dataset

The Debsources Dataset is a polished version of the data underpinning `sources.d.n`, suitable for a wide range of large-scale analyses on FOSS components released as part of Debian. The dataset is composed of two parts that can be used either together or independently:

Source code The dataset includes the source code of 10 Debian stable releases published over the past 2 decades, corresponding to 82 thousand packages for more than 30 million source code files. To reduce storage size, source code files have been deduplicated and organized in a manner that facilitates and speeds up empirical studies. The result of deduplication is 15 million unique files, requiring $\approx$ 320GB of disk space. After compression with `xz` the source code part of the dataset shrinks down to $\approx$ 90GB.

Metadata Rich metadata regarding all shipped source code are also part of the dataset. Release metadata link together the 10 Debian releases, the packages that compose each of them, and the source code files that form each package. In addition to release information, the dataset also contains the following content-oriented metadata:

[5] https://tracker.debian.org

- per-file size metrics including file size in bytes, number of lines (using `wc`), source lines of code (SLOC) divided by language, computed using both `sloccount`,[6] and `cloc`;[7]
- checksums: cryptographic hashes SHA1 [41] and SHA256, as well as locality-sensitive TLSH [120] hashes of all files;
- MIME media type of each file, as detected by `file`;[8]
- location, name, and type of developer-defined symbols (functions, data types, classes, methods, etc.) obtained indexing all source code with Exuberant Ctags;[9]
- applicable FOSS license for individual files, as detected by both `ninka` [73] and `fossology` [79].

Source code is shipped as a set of tarballs, metadata as a PostgreSQL [149] database dump.

4.2.1 Exploitation ideas

The Debsources Dataset is a valuable resource for scholars interested in studying either the composition or the long-term evolution of FOSS. Here are a few ideas—some already realized, some still up for grabs—on how to exploit the dataset:

- Conduct or replicate long-term, macro-level *evolution studies* of FOSS. We show an example of this in the case study of Section 4.3.

  Several followup research questions to that study remain unanswered, e.g.: does the use of different programming languages evolve in similar ways along the history of development? 20 years are enough to observe the raise and fall of programming languages and try to spot interesting adoption patterns. Using the Debsources Dataset those studies can be done both in aggregate ways (e.g., how many software projects are written in a given language over time?) and at per-project level (e.g., do all software projects written in a given language follow similar evolution patterns?).

- Study the structure and evolution of *license use* in FOSS, at different granularities: file, package, distribution. We address some of these in Section 4.3.4, but a lot remains to be done, most notably in the area of licensing of software components as aggregate wholes.

- Investigate *code reuse and cloning* along the whole history of all software packages contained in the dataset. Reuse without modification is trivial to track thanks to SHA1 and SHA256 checksums. Reuse *with* modification can be supported using ctags and/or TLSH hashes as fingerprinting techniques to track (modified) code copies, or by directly parsing the actual source code available in the dataset.

- The availability of source code can be further leveraged to support several kinds of static analysis studies. By focusing on source code files written in a specific programming language (e.g., C, C++), researchers can study the evolution over time

[6] http://www.dwheeler.com/sloccount/
[7] https://github.com/AlDanial/cloc
[8] http://www.darwinsys.com/file/
[9] http://ctags.sourceforge.net/

of bugs that are detectable with a given static analysis tool (e.g., Coccinelle [125], Coverity [23]).

On the more practical side, the source code in the dataset also forms an interesting benchmark for *code search* at a scale. Multi-language, license-aware, automatic code completion backed by the Debsources Dataset would make for a very fun and useful toy for many developers.

- In comparison with other sub-fields, *release engineering* [11] is still relatively unexplored in empirical software engineering. The Debsources Dataset allows to follow the evolution of package-level structures along 20 years of Debian, and to mix-and-match with release metadata, metrics, and actual source code. Some open research questions in this area are: when and how software projects get split into multiple packages? does package organization change over time? does that affect release schedules? how do packages migrate from one development release to another? etc.

4.2.2 Availability and reproducibility

The Debsources Dataset is Open Data. The metadata part of the dataset is available under the terms of the Creative Commons Attribution-ShareAlike (CC BY-SA) license, version 4.0. The source code part of the dataset is available under the terms of the applicable FOSS licenses. The dataset is available for download from Zenodo[10] at `https://zenodo.org/record/61089`, with DOI reference `10.5281/zenodo.61089`.

The Debsources Dataset has been assembled by mirroring and extracting Debian source releases, organizing extracted source code to remove duplicates, running analysis tools over the obtained files, and injecting their results in a PostgreSQL database.

Given that both all Debian releases and the analysis tools we have used are freely available as FOSS, the dataset can be recreated from scratch by anyone; a detailed, reproducible blueprint to do so is given in [30]. Be warned though that (re-)creating the dataset takes a significant amount of resources, both in terms of processing time and required disk space. The dataset (re-)creation process is I/O-bound and might require up to 1.5TB of working disk space during processing. In total, a realistic estimate for recreating from scratch the dataset on a consumer machine equipped with fast SSD drives is in between 4 and 5 weeks; a couple of weeks more with spinning disks. Note that to simply *use* the Debsources Dataset there is not need to *recreate* the dataset. The blueprint for doing so has been documented only for information and reproducibility purposes.

4.2.3 Source code

The first part of the Debsources Dataset is a set of 16 tarballs, each one weighting 5–6GB, containing the deduplicated source code files. Files contained in these tarballs are "sharded" into sub-directories; for example, a file whose SHA1 is `deadbeef[...]` will have path `de/ad/deadbeef[...]`. The additional tarball `debsources-ext.tar.xz` contains file extension information as a set of symlinks to the actual source code file.

[10] `https://zenodo.org/`

Table 4.2: Various versions of the `xournal` package in Debian, all containing a file named `src/xo-interface.h` with the same SHA1 checksum.

Release	Package	Version	SHA1 of `src/xo-interface.h`
lenny	xournal	0.4.2.1-0.1	e09a07941a3c92140c994fcdda7f74bce1af4ca3
squeeze	xournal	0.4.5-2	e09a07941a3c92140c994fcdda7f74bce1af4ca3
wheezy	xournal	0.4.6~pre20110721-1	e09a07941a3c92140c994fcdda7f74bce1af4ca3
jessie	xournal	1:0.4.8-1	e09a07941a3c92140c994fcdda7f74bce1af4ca3

Consider file `src/xo-interface.h`, which can be found in four different versions of the `xournal` package in Debian, as shown in Table 4.2. After extraction source code can be found in two top-level directories: `debsources` and `debsources.ext`. The first one contains the actual source code, the second extensionful symlinks to them. In our example we will have the following on-disk layout (names ending in '/' represent directories, and '->' a symlink and its destination):

```
debsources/
...
debsources/e0/
debsources/e0/9a/
debsources/e0/9a/e09a07941a3c92140c994fcdda7f74bce1af4ca3
...
debsources.ext/
...
debsources.ext/e0/
debsources.ext/e0/9a/
debsources.ext/e0/9a/e09a07941a3c92140c994fcdda7f74bce1af4ca3.h ->
   ../../../sources/e0/9a/e09a07941a3c92140c994fcdda7f74bce1af4ca3
...
```

Even though files are renamed to match their SHA1 checksums, the directory structure of individual packages is not lost. Full original paths are available as part of the metadata database described next. As a preview, the following SQL query can be used to reconstruct the paths at which a file with the SHA1 of our example can be found in the dataset, together with the corresponding package names and versions:

```
SELECT release_id as release,
       package_name as package, package_version as version,
       encode(path, 'escape') as path
FROM releases
NATURAL JOIN package_info
NATURAL JOIN paths
NATURAL JOIN path_info
NATURAL JOIN files
WHERE sha1='e09a07941a3c92140c994fcdda7f74bce1af4ca3'
```

When run the query will return the following tuples:

Table 4.3: Tools used to extract file-level metadata.

Tool	Version	Command
file	5.14	`file -mime-type`
cloc	1.66	`cloc -by-file -follow-links -skip-uniqueness \ -sql-append`
sloccount	2.26	`sloccount -duplicates -follow -details`
wc (coreutils)	8.13	`wc -l`

release	package	version	path
lenny	xournal	0.4.2.1-0.1	src/xo-interface.h
squeeze	xournal	0.4.5-2	src/xo-interface.h
wheezy	xournal	0.4.6~pre20110721-1	src/xo-interface.h
jessie	xournal	1:0.4.8-1	src/xo-interface.h

4.2.4 Metadata

The second part of the dataset is a Postgres database, containing all source code metadata. The database schema is shown in Figure 4.3. A brief description of each table is given below.

Intrinsic information

The following tables describe releases, packages, files and the relationships among them:

package_info information about Debian source packages contained in the dataset, such as package names, versions, and associated attributes (e.g., project homepage). Package names are generally lowercase variants of the original (or "upstream") FOSS project names, e.g., bash, linux (the kernel), libreoffice, etc. Package versions include both the upstream project version and the Debian package revision separated by a dash, e.g.: "1.2.3-4".

release_info information about the 10 Debian releases in the dataset; name, version, and release date of each one are included.

releases mappings between source packages and Debian releases.

path_info the full path name of every file in the dataset. The table also contains, as a separate field, the file extension.

files deduplicated files together with their checksums (SHA1, SHA256, and TLSH) and size (in bytes). This table lists all unique files present in the dataset.

paths ternary mappings between *packages*, *path_info*, and *files*. This table indicates, for a given package and path, the corresponding unique file. For symbolic links, the *file_id* column will be NULL.

Derived information

The following tables describe information that have been extracted from Debian source code using a variety of indexing and measurement tools:

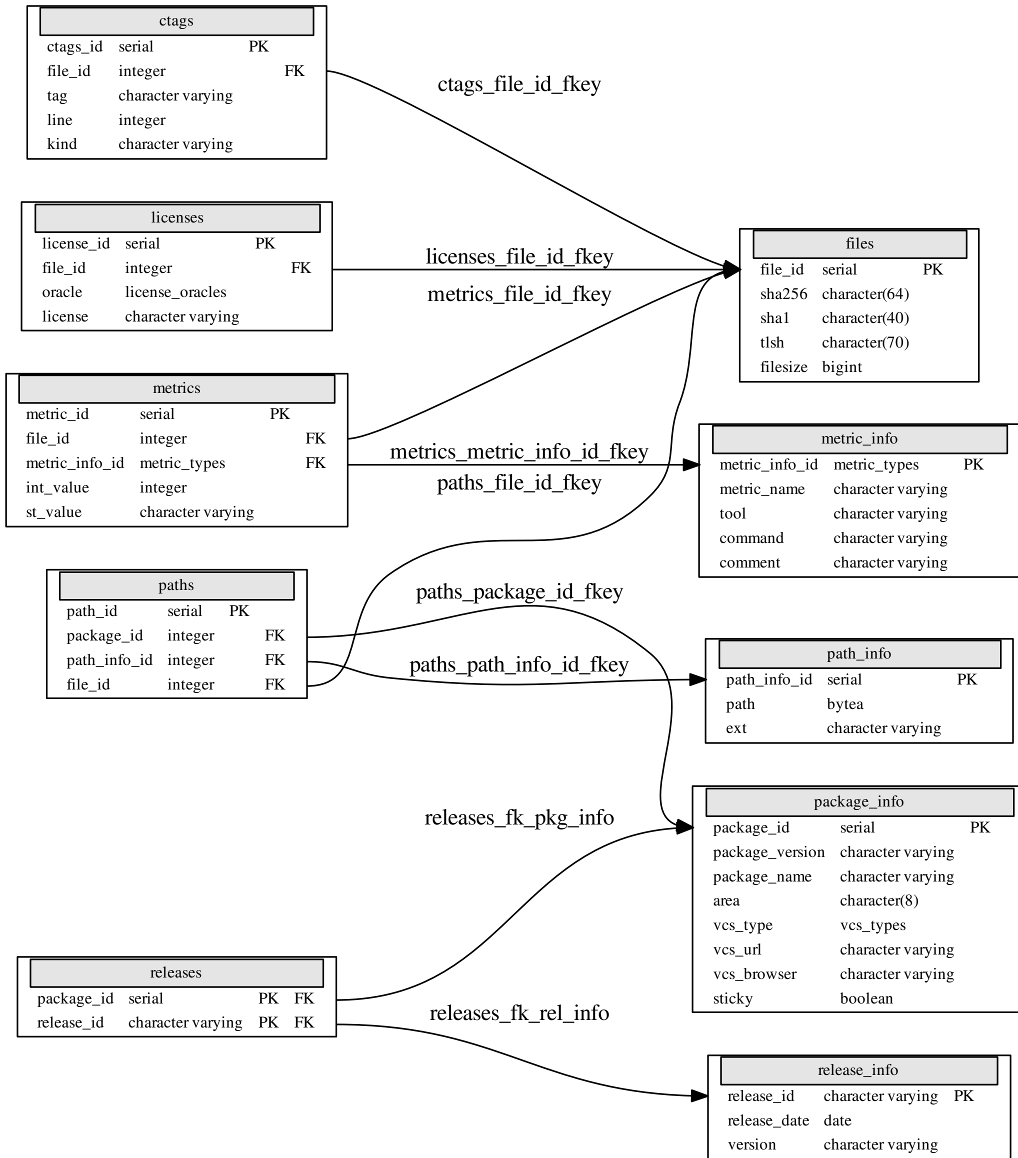


Figure 4.3: Database schema. Primary key fields are denoted with “PK”, foreign keys “FK”; arrows indicate referential integrity constraints.

Table 4.4: Size of Debsources Dataset metadata as a Postgres database. The entire database requires ≈ 40GB of disk space (including indexes, which are not listed below).

Table	Disk size	Tuples
ctags	23 GB	186.5M
files	5944 MB	15.5M
metrics	3549 MB	46.7M
paths	3259 MB	30.5M
licenses	2976 MB	31.0M
path_info	1895 MB	11.7M
package_info	14 MB	82113
releases	7248 KB	97471
metric_info	32 KB	4
release_info	32 KB	10

licenses license information. This table maps unique files to the corresponding FOSS licenses as identified by the license detection tools (or "oracles") `ninka` and `fossology`.

metric_info information about the tools used to compute file-level information. For reproducibility reasons, this table includes tool name, version, command line used to run it, and a *comment* field with additional human-readable information. Due to how representative of the available file-level information this table is, its content is also shown in Table 4.3.

metrics links files to the information extracted from them. Some derived information are integer-valued (e.g., the output of `wc -l`), some string-valued (e.g., `file` output), some both (`cloc` and `sloccount` output both detected language and number of SLOCs). For this reason this table allows to store an integer (field `int_value`) and/or a string (`st_value`). The attribute `comment` in table *metrics_info* documents which field is relevant for which metric.

ctags `ctags` results for each file. The table contains one entry for each developer-defined symbols in a given source file, together with the precise file location at which the symbol was found and the symbol type (function, data type, method, etc).

4.2.5 Dataset size

To give an idea of the size of the dataset, Table 4.4 lists the sizes of all tables in the database, as both number of tuples and required disk space. If space is at a premium, some large tables (e.g., *ctags*) can be skipped without compromising the referential in-

Table 4.5: Size of the source code part of the Debsources Dataset.

Tarball	Disk usage (compressed)	Disk usage (expanded)
`debsources.*.tar.xz`	89GB (total)	317GB
`debsources-ext.tar.xz`	422MB	61GB
`debsources.dump.xz`	3.1GB	see Table 4.4

tegrity of the database. Similarly, Table 4.5 details the required disk space to locally host the source code part of the Debsources Dataset.

4.3 Case study: long-term macro-level evolution

In the remainder of this chapter we show how the Debsources Dataset can be used to conduct a long-term, macro-level evolution analysis of FOSS projects, as they can be observed through the lens of the Debian distribution. We focus on aspects such as source code size (under various metrics), programming language popularity, package size, package maintenance, and software licensing.

The analyses we conduct are both qualitative and quantitative, and in part replicate and extend previous findings [80, 31]. The research questions we will address are:

RQ i. *How does the size of Debian evolve over time?* Looking at various metrics we will study how and at which rate Debian grows across releases.

RQ ii. *How much Debian changes between releases?* By studying package versions and their content, we can measure the amount of packages that are updated across Debian releases and to what extent they are.

RQ iii. *How has the popularity of programming languages changed over the last 20 years?* By looking at the evolution of SLOCs per language, we identify which languages are gaining (or losing) traction among FOSS projects represented in Debian.

RQ iv. *Which licenses apply to Debian source code files?* We identify which software licenses are used in Debian at a file-by-file granularity, irrespectively of the containing package.

RQ v. *Which licenses can be found in Debian source packages?* By aggregating file licenses by package we can study the expected license variability when reusing entire software packages.

RQ vi. *How has license use evolved in Debian over time?* We explore the evolution of license use over time by comparing the licensing of files and packages that belong to different Debian releases.

4.3.1 Growth over time

The evolution of Debian size over time (RQ i) can be studied under various metrics. We take into account the following ones: number of packages, number of source code files, disk usage of (uncompressed) source code, lines of code (SLOCs), and developer-defined symbols (or "*ctags*").

In the Debsources Dataset packages can be found in the *package_info* table, that has one row per package. Source code files can be found in table *paths*, which in turn points to unique files listed in table *files*. All file-level metrics, except ctags, are in table *metrics*, column *int_value*, distinguished by metric type (column *metric_info_id*).[11] File-level metrics can then be grouped by package following the *metrics* → *files* ↔ *paths* → *package_info*

[11] Note that two different SLOC metrics are available in the dataset: as computed by `sloccount` and `cloc`. Each tool has its strength and weaknesses. For this case study we use `sloccount` numbers.

Table 4.6: Debian release sizes by various metrics—number of packages, files (and files explicitly recognized as source code by `sloccount`), disk usage of uncompressed source packages, lines of code, developer-defined symbols (ctags). See also Table 4.7 for additional statistics parameters about these measures.

Release	Version	Packages	Files (k)	Source files (k)	Disk usage (GB)	ctags (M)	SLOCs (M)
hamm	2.0	1373	348.4	152.5	4.1	4.1	34.9
slink	2.1	1880	484.6	224.4	6.0	6.2	51.9
potato	2.2	2962	686.0	292.6	8.6	7.4	68.8
woody	3.0	5583	1394.5	563.3	18.2	17.2	140.7
sarge	3.1	9050	2394.0	870.6	34.1	24.2	210.1
etch	4.0	10 550	2879.7	1092.7	45.0	30.3	272.1
lenny	5.0	12 517	3713.9	1437.2	61.8	38.3	332.7
squeeze	6.0	14 951	4908.1	1952.2	89.1	52.3	444.4
wheezy	7	17 564	7310.5	2751.4	131.7	69.5	636.8
jessie	8	21 041	8375.0	3404.2	167.0	95.6	784.3

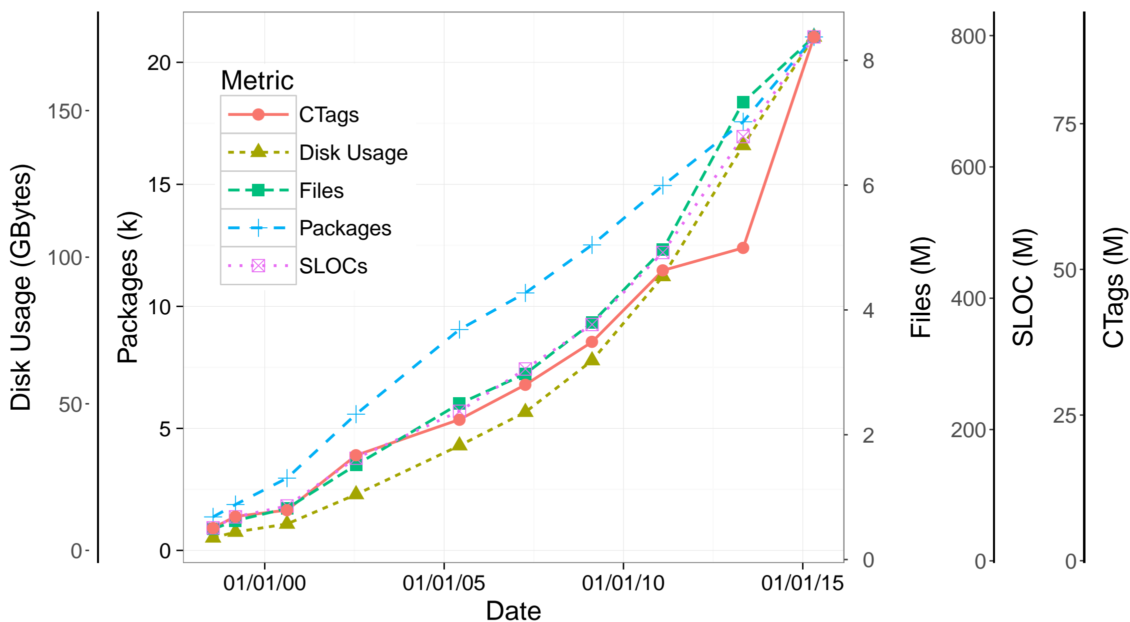


Figure 4.4: Debian release size over time, under various metrics.

chain of relationships. Ctags are stored in the separate *ctags* table because, whereas they can be used as a size/complexity metric for individual source code files, they primarily act as an index which doesn't fit the general model of the *metrics* table. Per-packages metrics can be further aggregated by release using the *releases* table. Per-release metrics can finally be sorted by time using the *release_date* field of the *release_info* table.

Table 4.7: Averages, median, and maximum of various size metrics, over packages and per release. See Table 4.6 for totals.

	Files		Disk usage (KB)		SLOCs		
Release	**median**	**max**	**median**	**max**	**median (K)**	**avg. (K)**	**max (M)**
hamm	65	17.8	780	0.2	4.63	25.4	1.2
slink	64	17.8	782	0.1	4.37	27.6	1.3
potato	58	27.3	732	0.2	3.46	23.2	2.0
woody	60	29.8	784	0.4	3.61	25.2	2.9
sarge	62	68.6	904	0.9	3.74	23.2	4.0
etch	65	27.2	1012	0.4	4.54	25.8	5.6
lenny	66	59.6	1000	0.9	4.41	26.5	5.9
squeeze	69	57.2	960	2.3	4.17	29.7	7.9
wheezy	69	182.4	924	2.8	3.97	36.2	13.9
jessie	67	182.4	808	2.8	3.40	37.3	14.9

Release size

The above query plan can easily be translated to SQL queries and run on the Debsources Dataset. Query results are shown in Table 4.6, and plotted in Figure 4.4 over time.

In absolute terms, Debian has scaled to a point where the Jessie stable release (2015) contains more than 21 thousand packages, and almost 800 million lines of code. If we look at the metrics evolution over time we notice that the five considered metrics exhibit similar growth rates. Four of them (ctags, disk usage, files, and SLOCs) are very highly correlated and grow super-linearly, with an apparent slow down in the most recent stable release. The other metric (package count) is more regular and almost perfectly linear.

This discrepancy gives some insights about Debian technical management. Packages are the units at which software is maintained in Debian: each package is under the responsibility of a (group of) maintainer(s). A super-linear growth in the number of packages would need a super-linear growth in the number of maintainers to be sustainable in the long-term or, alternatively, an increase in the amount of packages maintained by the same people. While there is some evidence of the latter [133] on shorter time-frames (about a decade) than the one considered here (two decades), it also seems that Debian is focusing on sustainable size increases rather than trying to package every available FOSS product bearing the risk of stretching its forces too thin.

Package size

Thanks to the mapping between metrics and packages, we can also study the distribution of package sizes in different Debian releases: it is plotted in Figure 4.5 for selected releases. Averages, medians, and maximums of selected metrics over packages are given in Table 4.7.

Increasingly, more and more very large packages are present in Debian: in Jessie the

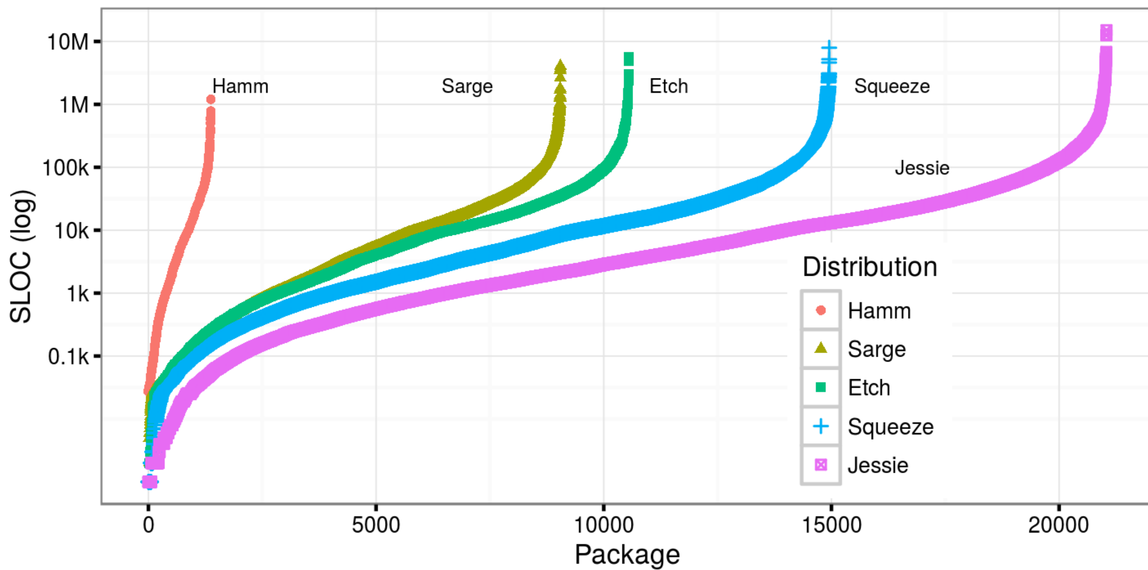


Figure 4.5: Size of packages per distribution (measured in SLOC, y-axis). Each integer in the x-axis represents one package. E.g., in Jessie ≈ 7500 packages have sizes less or equal to 1k SLOC, while ≈ 20 000 packages have sizes less or equal to 100k SLOC.

chromium-browser and *linux* packages have, respectively, more than 15M and 12M SLOC. When Hamm was released its biggest package was *xfree86*, with "only" 1.2M SLOC. At the same time the per-release averages of package size are going up, whereas medians are going down. Overall it appears that: i) smaller and smaller packages are getting added to Debian, ii) larger and larger packages are getting added too; with (ii) dominating more and more the total size of releases.

A possible explanation for (i) comes from the packaging of relatively new software ecosystems that are increasingly releasing very small packages, e.g., Python's PyPi, R's CRAN, Node.js' NPM, etc. (ii) on the other hand seems due to behemoth software packages such as Web browsers, that are becoming self-contained work environments that need to (re)implement more, and more complex, functionalities that were historically available from separate packages.

4.3.2 Package maintenance

RQ ii is about the amount of changes that Debian users can expect when upgrading from one stable release to another. As the pairs ⟨name, version⟩ uniquely identify packages throughout Debian history, and as those pairs are available in the *package_info* table, we can leverage the Debsources Dataset to compare the sets of packages shipped by different Debian releases. Furthermore we can dissect package versions into their upstream and Debian-specific parts, separating the debian-specific part of package versions from the upstream one, to relate changes in the Debian archive to upstream ones.

The top-half of Table 4.8 summarizes the amount of changes between pairs of Debian releases. *Common* packages are those that appear in both releases, in the same or dif-

Table 4.8: 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
hamm	1324c 842u	1198c 463u	1079c 270u	958c 175u	864c 148u	782c 124u	719c 100u	670c 81u	649c 73u
slink		1657c 742u	1455c 384u	1281c 252u	1155c 210u	1037c 172u	941c 136u	881c 113u	852c 101u
potato			2456c 935u	2118c 551u	1881c 436u	1683c 352u	1497c 271u	1399c 220u	1348c 201u
woody				4588c 1688u	3953c 1156u	3497c 908u	3018c 633u	2786c 520u	2648c 458u
sarge					7671c 3832u	6828c 2597u	5896c 1717u	5349c 1367u	5042c 1164u
etch						9230c 4578u	8033c 2906u	7212c 2203u	6778c 1813u
lenny							10823c 5271u	9624c 3673u	8999c 2928u
squeeze								13098c 6802u	12201c 4890u
wheezy									16160c 8427u

	from previous suite to								
	slink	**potato**	**woody**	**sarge**	**etch**	**lenny**	**squeeze**	**wheezy**	**jessie**
modified pkgs	556m	1305m	3127m	4462m	2879m	3287m	4128m	4466m	4881m
changed files per pkg	54.6%	64.4%	65.3%	67.5%	58.9%	59.8%	60.4%	57.3%	54.7%

ferent versions. *Unchanged* packages appear in both releases with the same "upstream" version, ignoring Debian-specific version changes (hence: *unchanged* ⊆ *common*).

It is interesting to note that 73 packages have remained at the same upstream version between Hamm and Jessie, for more than 17 years, whereas their Debian revisions have evolved. Among these packages we can find for instance *netcat*, a network tool that hasn't changed upstream for that long, but seems to be still working just fine in Debian (otherwise it would have been removed from recent releases). This hints at the fact that long lasting unchanged packages might have been abandoned upstream, but are still maintained in Debian via patches applied by distribution maintainers.

The bottom-half of Table 4.8 focuses on upgrades from any given release n to the immediately subsequent release $n+1$, which is the most common upgrade path in Debian. The table shows the number of *modified* packages between consecutive releases (packages which exist in both releases, but in different upstream versions), as well as the proportion of source code files updated in these packages. The latter can be computed using the already discussed mapping between files and packages, together with either SHA1 or SHA256 checksums, both available in the *files* table.

The percentage of common and unchanged packages with respect to the previous release oscillates around 87% (common) and 43% (unchanged) with low variance. This suggests that Debian users experience high stability in terms of which packages are available across releases (almost 90%), as well as a steady flow (around 60%) of new upstream releases that are incorporated by Debian maintainers. The number of changed files per package on the other hand gives insights into how much new upstream releases touch the actual source code that form packages. This measure is also pretty stable across all Debian releases, ranging between 54% and 67%. Note however that this does not tell us how much individual files have been changed, only how many of them have: bumping copyright year in a file header or rewriting the file from scratch will still account for one source file change. More precise evaluations of "how much" source code has changed can be performed leveraging TLSH hashes, that are readily available in the Debsources Dataset as well.

4.3.3 Programming language popularity

To address RQ iii (programming language popularity) we can simply aggregate per-package first, and per-release then, the SLOC counts available in the *metrics*

The evolution of programming languages in Debian, as computed by `sloccount`, is presented in Table 4.9 and plotted in Figure 4.6 and 4.7. In both cases we restrict presented results to the most popular languages, using the Jessie release as a reference. Figure 4.6 shows the evolution of language popularity in absolute SLOCs, while Figure 4.7 shows the proportion over release size measured in SLOC.

Results show that C has always been and still is the dominant language in Debian, since a big part of the core operating system (the Linux kernel, the GNU suite, etc) is written in C. However, while the absolute amount of C code has been steadily increasing, its proportion over the total has been decreasing since Slink (1999). Other languages, and most notably C++, are getting more and more relevant. The proportion of C code seems to have been stable for the past 3 releases though, at about 41% of the total.

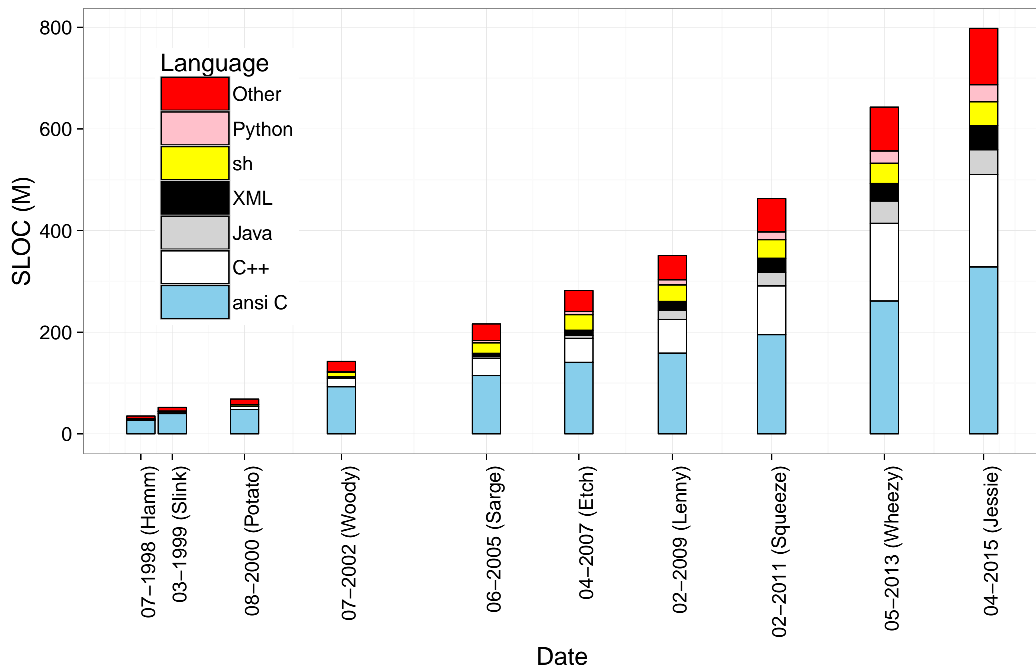


Figure 4.6: Evolution of the most popular (top-6 plus other) programming languages in Debian by total number of SLOC per release.

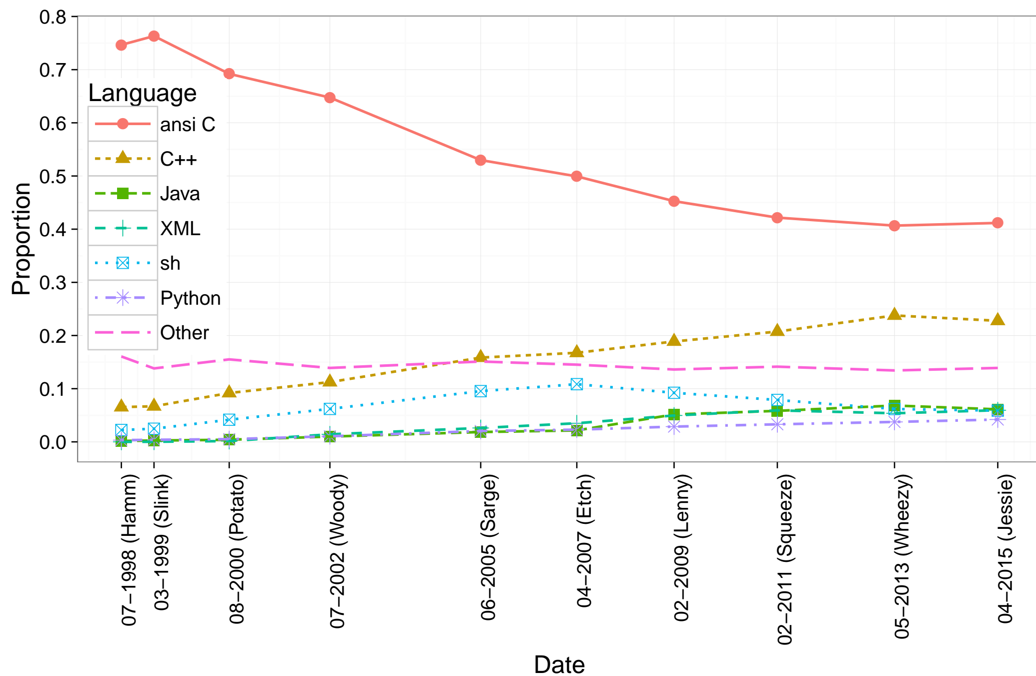


Figure 4.7: Evolution of the most popular (top-6 plus other) programming languages in Debian as a proportion of release size in SLOC.

Table 4.9: Most popular programming languages in Debian releases, in MSloc. Numbers between parentheses represent percentage of total. (Quantities < 0.1 have been omitted and replaced by ≈ 0.)

Release	**total**	**ada**	**ansic**	**asm**	**cpp**	**erlang**
hamm	35	0.24 (0.68)	27 (77)	0.39 (1.1)	1.9 (5.5)	NA (NA)
slink	52	0.26 (0.51)	4.5 (78)	0.64 (1.2)	3.0 (5.7)	NA (NA)
potato	69	0.42 (0.61)	49 (7.6)	0.57 (0.83)	5.8 (8.5)	0.21 (0.30)
woody	15	0.58 (0.41)	94 (67)	2.6 (1.9)	15 (1.4)	≈ 0 (≈ 0)
sarge	21	1.1 (0.53)	120 (56)	2.8 (1.3)	33 (16)	≈ 0 (≈ 0)
etch	270	0.76 (0.28)	140 (53)	4.5 (1.6)	46 (17)	0.69 (0.25)
lenny	330	0.85 (0.26)	160 (49)	4.1 (1.2)	64 (19)	0.82 (0.25)
squeeze	440	1.3 (0.29)	210 (46)	4.8 (1.1)	96 (22)	1.3 (0.28)
wheezy	640	1.6 (0.25)	290 (46)	8.2 (1.3)	150 (23)	1.6 (0.25)
jessie	780	1.8 (0.23)	360 (46)	1.5 (1.3)	180 (23)	1.8 (0.23)

Release	**f90**	**fortran**	**haskell**	**java**	**lisp**
hamm	≈ 0 (≈ 0)	0.70 (2.0)	NA (NA)	≈ 0 (0.17)	0.11 (0.32)
slink	≈ 0 (≈ 0)	1.0 (2.0)	≈ 0 (≈ 0)	0.13 (0.25)	2.5 (4.8)
potato	≈ 0 (≈ 0)	1.4 (2.1)	≈ 0 (≈ 0)	0.27 (0.40)	3.4 (4.9)
woody	≈ 0 (≈ 0)	2.3 (1.6)	0.28 (0.20)	1.4 (1.0)	5.1 (3.7)
sarge	≈ 0 (≈ 0)	2.9 (1.4)	0.98 (0.47)	4.0 (1.9)	6.9 (3.3)
etch	≈ 0 (≈ 0)	2.1 (0.76)	0.58 (0.21)	6.1 (2.2)	7.2 (2.6)
lenny	0.29 (≈ 0)	2.3 (0.68)	0.67 (0.20)	18 (5.4)	8.1 (2.4)
squeeze	0.76 (0.17)	2.5 (0.56)	0.93 (0.21)	27 (6.1)	9.7 (2.2)
wheezy	1.1 (0.17)	8.2 (1.3)	1.6 (0.25)	44 (7.0)	8.8 (1.4)
jessie	7.5 (0.95)	9.7 (1.2)	2.0 (0.25)	50 (6.3)	11 (1.4)

Release	**makefile**	**ml**	**objc**	**pascal**	**perl**	**python**
hamm	2.3 (6.7)	≈ 0 (0.26)	≈ 0 (0.16)	0.17 (0.49)	≈ 0 (≈ 0)	0.49 (1.4)
slink	0.15 (0.28)	≈ 0 (0.11)	0.22 (0.43)	≈ 0 (0.10)	0.79 (1.5)	0.20 (0.39)
potato	0.21 (0.31)	0.15 (0.22)	0.41 (0.60)	0.31 (0.45)	1.4 (2.0)	0.36 (0.52)
woody	0.37 (0.26)	0.38 (0.27)	0.55 (0.39)	0.43 (0.31)	3.0 (2.1)	1.5 (1.1)
sarge	0.55 (0.26)	0.76 (0.36)	0.76 (0.36)	1.4 (0.65)	6.3 (3.0)	4.4 (2.1)
etch	0.68 (0.25)	1.3 (0.47)	1.0 (0.37)	1.1 (0.41)	8.1 (3.0)	6.5 (2.4)
lenny	0.75 (0.23)	1.8 (0.55)	1.1 (0.32)	0.87 (0.26)	9.4 (2.8)	1.1 (3.0)
squeeze	0.69 (0.16)	2.6 (0.59)	1.2 (0.27)	3.8 (0.84)	13 (2.9)	16 (3.5)
wheezy	0.66 (0.10)	3.8 (0.59)	1.7 (0.26)	4.4 (0.69)	18 (2.8)	25 (3.9)
jessie	0.72 (≈ 0)	4.1 (0.52)	1.9 (0.25)	5.5 (0.70)	20 (2.5)	35 (4.5)

Release	**php**	**ruby**	**sh**	**sql**	**tcl**	**yacc**
hamm	≈ 0 (≈ 0)	≈ 0 (≈ 0)	0.92 (2.6)	≈ 0 (≈ 0)	0.35 (1.0)	0.19 (0.54)
slink	≈ 0 (≈ 0)	≈ 0 (≈ 0)	1.5 (2.9)	≈ 0 (≈ 0)	0.50 (0.97)	0.25 (0.48)
potato	≈ 0 (≈ 0)	≈ 0 (≈ 0)	3.3 (4.8)	≈ 0 (≈ 0)	0.67 (0.97)	0.32 (0.47)
woody	0.58 (0.41)	≈ 0 (≈ 0)	9.5 (6.8)	≈ 0 (≈ 0)	1.2 (0.86)	0.45 (0.32)
sarge	1.8 (0.87)	0.46 (0.22)	21 (1.2)	≈ 0 (≈ 0)	2.1 (1.0)	0.56 (0.26)
etch	3.0 (1.1)	1.2 (0.45)	31 (12)	0.51 (0.19)	1.7 (0.64)	0.65 (0.24)
lenny	4.0 (1.2)	2.0 (0.61)	33 (9.9)	0.66 (0.20)	1.9 (0.58)	0.67 (0.20)
squeeze	4.7 (1.1)	4.3 (0.96)	38 (8.6)	1.5 (0.34)	2.5 (0.55)	0.81 (0.18)
wheezy	5.8 (0.92)	4.2 (0.66)	42 (6.6)	2.4 (0.38)	2.6 (0.41)	1.0 (0.16)
jessie	8.1 (1.0)	5.2 (0.66)	49 (6.2)	3.9 (0.49)	3.1 (0.40)	1.2 (0.15)

Table 4.10: Median file size (in SLOC) per language for the most popular languages.

Release	ada	ansic	asm	cpp	erlang	f90	fortran	haskell	java	lex	lisp
hamm	40	69	26	62	-	47	67	-	44	180	130
slink	38	68	43	60	-	11	61	12	36	190	120
potato	40	71	34	57	170	11	73	44	34	170	120
woody	47	86	75	64	42	16	81	36	37	210	140
sarge	47	79	40	66	43	38	89	37	43	180	120
etch	43	80	32	64	210	61	79	57	46	170	120
lenny	42	79	25	61	200	89	78	44	44	180	130
squeeze	45	76	20	62	160	96	76	58	45	190	120
wheezy	47	80	23	66	130	98	110	32	47	190	110
jessie	38	75	21	65	110	79	96	39	47	210	110

Release	make	ml	objc	pascal	perl	php	python	ruby	sh	sql	tcl	yacc
hamm	43	26	150	140	62	≈ 0	59	≈ 0	20	19	87	520
slink	42	22	120	12	66	≈ 0	61	≈ 0	23	16	92	510
potato	42	32	140	63	63	19	68	31	23	17	97	600
woody	46	35	170	72	61	45	65	31	27	16	80	320
sarge	47	44	160	92	63	39	60	38	35	11	94	320
etch	47	50	150	300	70	46	59	45	38	11	92	320
lenny	44	49	150	230	64	46	58	42	39	11	84	320
squeeze	32	50	140	84	56	44	59	38	37	26	69	320
wheezy	15	48	130	79	58	39	60	36	33	20	72	340
jessie	11	43	110	88	53	37	62	32	34	20	65	350

Without a comprehensive reference base for FOSS source code as a whole[12] it is impossible to determine how representative these numbers are of out-of-Debian trends. But the comparison with programming languages trends on other platforms (e.g., GitHub [101]) is striking. Whereas GitHub developers seem to be flocking to JavaScript, Java, Ruby, and PHP, a foundational operating system like Debian is still prominently composed of system-level languages like C and C++.

File size

We can drill down to investigate median file sizes (in SLOC) per language and their evolution over time. Detecting the programming language of each source file can be done in a number of ways; in the following we have relied on language detection as performed by `sloccount`.

Table 4.10 presents per release and per language median file sizes for the most popular languages. For the top-6 of them, their evolution over time is plotted in Figure 4.8.

Most of the studied languages are shown to be relatively stable in their median file size over time. This is the case for mainstream languages such as C, C++, and Java, as well as several others such as Perl and Lisp (not plotted). Median file size also appears to be a rather intrinsic characteristic of a programming language, that is not really affected by how popular the language is in a large FOSS ecosystem.

[12] we will get back to this problem in Chapter 5, as it is something that Software Heritage can help addressing

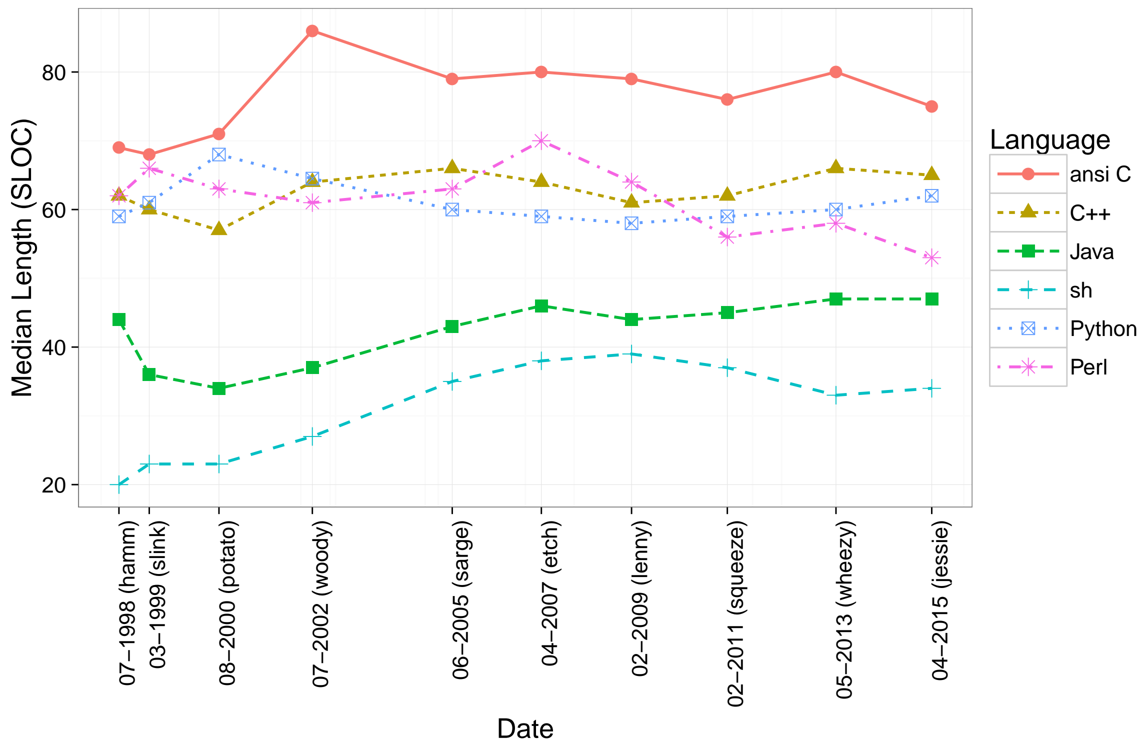


Figure 4.8: Evolution over time of the median file size (in SLOC) per language, based on file extension.

4.3.4 Debian licensing over time

One of the most important characteristics that define a FOSS component is its license. It is very important to know the license of a package, as it determines the rights and obligations of anybody wanting to reuse and further distribute the software (either as a component or as a stand-alone product). Since the conception of Debian in 1993 the FOSS license landscape has evolved significantly. Many licenses released new versions, others have been created, and some ceased to be used. The long history of Debian creates a perfect subject to evaluate how FOSS licenses use has evolved over time, and the popularity of licenses currently in use.

Creating a census of licenses used in a large software distribution is not an easy task though [99]. The first challenge is how to identify the licenses of individual files. Then, one needs to consider the overall licenses of aggregate/composite software bundles, such as packages in the case of FOSS distributions, which is not necessarily the same of the files that compose it. In the following we focus on the license of individual files. We do, however, aggregate file licenses by package in order to study license variability within packages, answering RQ v. Finally, in order to answer RQ vi (*how has license use evolved in Debian over time?*), we analyze the evolution of our answers to RQ iv (file licensing) and RQ v (source package licensing) across all Debian stable releases.

As automatic license identification of a file is still difficult and error prone, we avoid developing in house heuristics and rather resort to two state-of-the-art tools in license

Table 4.11: Number of different licenses identified in each release.

Release	Licenses
hamm	281
slink	326
potato	437
woody	620
sarge	949
etch	1125
lenny	1352
wheezy	1879
jessie	2039

identification—`ninka` [73] and `fossology` [79]—whose results when applied to the entire Debian corpus are readily available from the Debsources Dataset, in the *licenses* table. For the sake of brevity in the following we only discuss `fossology` results; also, we ignore the licenses of all files not recognized as being source code, such as binary files.

Individual file licensing

Table 4.11 shows the total amount of *different* licenses identified for each release in the dataset. As it can be observed, the number of identified licenses is very large and has grown an order of magnitude across Debian history. Part of the reason is that, when a file is licensed under two or more licenses, such combination of licenses is considered to be a different license by license identification tools. For example, in several Debian releases Firefox is licensed under a combination of the MPL, GPL-2.0, and LGPL-2.1. In most releases, few licenses account for most of the identified licenses: the top 50 most frequently identified licenses (including "No_license_found") correspond to 94–97% of release source files.

Table 4.12 shows the top identified licenses in the oldest and newest Debian releases available in the dataset. As it can be observed, most frequently files do not have a license that `fossology` can directly identify. Figure 4.9 shows the evolution over time of the most common identified licenses. As it can be seen, the most used licenses have been the GPL family, with recent increases for Apache-2.0 and the Mozilla Public License (MPL).

Package licensing and variability

In order to address RQ v (source package licensing), it is not practical to simply report each and every license found in every package. Instead, we develop several metrics, each one highlighting different aspects of the licensing of source code files belonging to a given package:

- How detectable are the licenses of the package source files? For this purpose we compute the proportion of files for which a license was identified over the total number of files. `fossology` reports *No_license_found* when it does not find the license of a given file, and *UnclassifiedLicense* when it finds one it does not know. Hence we consider a file to have an *identifiable license* if `fossology` reports a license

Table 4.12: Top identified licenses in two selected releases.

Release	License	Files	Prop.(%)	Accum. (%)
Hamm	No_license_found	72,533	47.5	47.5
	GPL-2.0+	22,983	15.1	62.6
	LGPL-2.0+	14,608	9.6	72.2
	BSD-style	3,667	2.4	74.6
	See-doc(OTHER)	2,490	1.6	76.2
	MIT-style	2,457	1.6	77.8
	UnclassifiedLicense	2,359	1.5	79.4
	GPL	2,329	1.5	80.9
	BSD-4-Clause-UC	2,112	1.4	82.3
	See-file	1,938	1.3	83.5
	X11	1,887	1.2	84.8
Jessie	No_license_found	1,011,088	29.7	29.7
	GPL-2.0+	432,482	12.7	42.4
	Apache-2.0	168,655	5.0	47.4
	GPL-3.0+	160,233	4.7	52.1
	GPL-2.0	148,364	4.4	56.4
	LGPL-2.1+	141,747	4.2	60.6
	BSD	115,135	3.4	64.0
	LGPL-2.0+	87,153	2.6	66.5
	BSD-3-Clause	72,634	2.1	68.7
	MIT	71,022	2.1	70.8
	EPL-1.0	66,755	2.0	72.7

other than *No_license_found* and *UnclassifiedLicense*. The *proportion of files without a license* is the number of source files without an identifiable license divided by the total number of source files.

- How many *different* licenses can be found in a given package? In this case we ignore files without an identifiable license.
- What is the *dominant license* identified in each package? We define such a license as the most commonly identifiable license, counted as the number of files it applies to. If two or more licenses are equally frequent, all of them are considered to be equally dominant.
- How much *diversity* is there in the licenses of the files of a package? The more licenses a package contains, the larger it will be the problem space of determining its license as an aggregate—due to how license compatibility works the problem will not necessarily be more *difficult*, but more options will have to be considered.

  To establish license diversity within a package we use Wilcox's Analog of the Mean Difference (MNDif). It represents the average of the absolute differences of all the possible pairs of license frequencies. Intuitively, it is the equivalent of a GINI coefficient, but applicable to categorical data. A value of 0 implies that all files have the same license, while a value of 1 that all licenses are equally represented, i.e., each license is used by the same number of files.

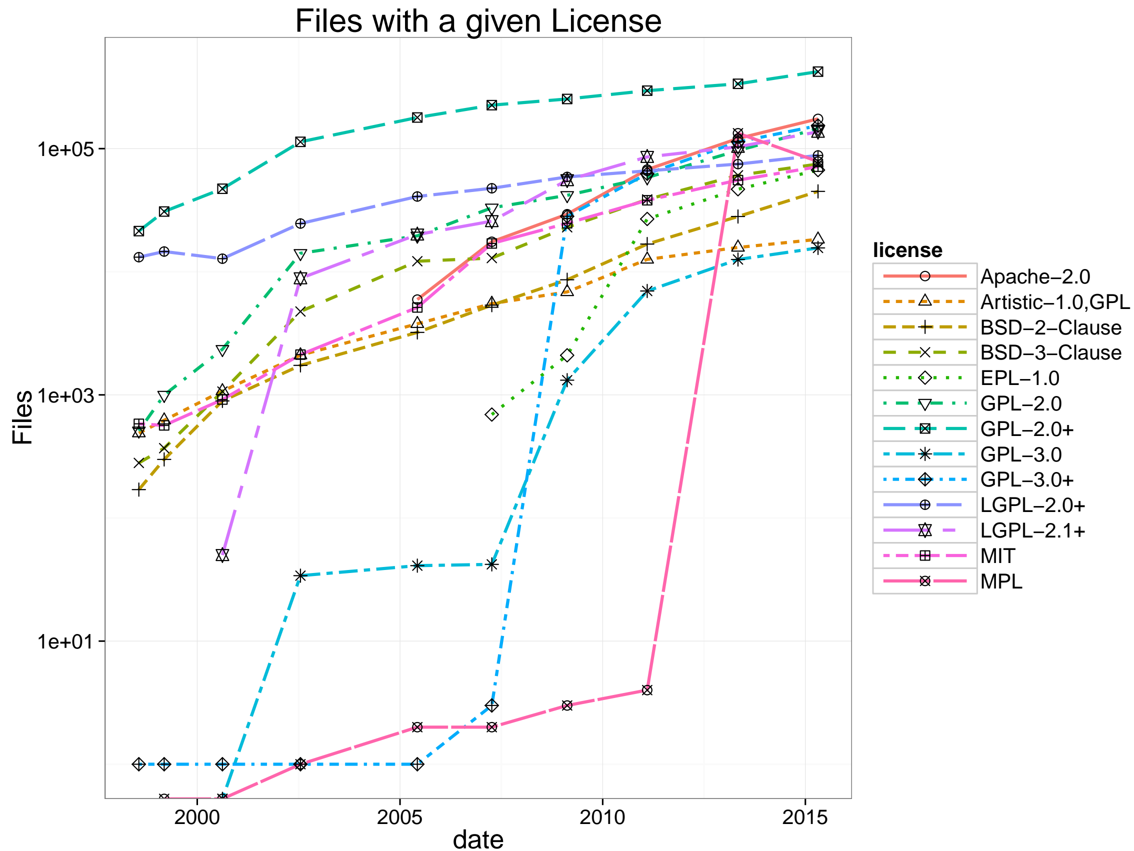


Figure 4.9: Evolution of the number of files with a given license, as detected by `fossology`.

When aggregating by package, different licensing patterns appear. Figure 4.10 shows box plots with the proportion of files that do not have a detectable license. The median number of source files without an identifiable license has fluctuated between 50% and 60%, showing the same pattern over time. It is important to mention that `sloccount` is relatively aggressive on what it considers source code. For example, `sloccount` considers Makefiles and configuration and installation scripts to be source code; these files do not normally include a license header. For this reason we also include the box plots for C (Figure 4.11) and Java files (Figure 4.12). As it can be seen, their current median is below 5% in both cases and, over time, the proportion of files without a license keeps dropping. It seems that, at least from the point of view of `fossology` and for mainstream programming languages, FOSS development practices (and in particular writing down license annotations) are evolving in a way that makes automatic license detection easier. There is still plenty of room for improvement though.

The number of licenses used per package is generally very small, as shown in Figure 4.13. The median is 2; the third quartile has decreased from 3 to 2 licenses in recent releases (taking into account identifiable licenses only).

With regard to the chosen licenses, we present in Figure 4.14 the evolution of licenses

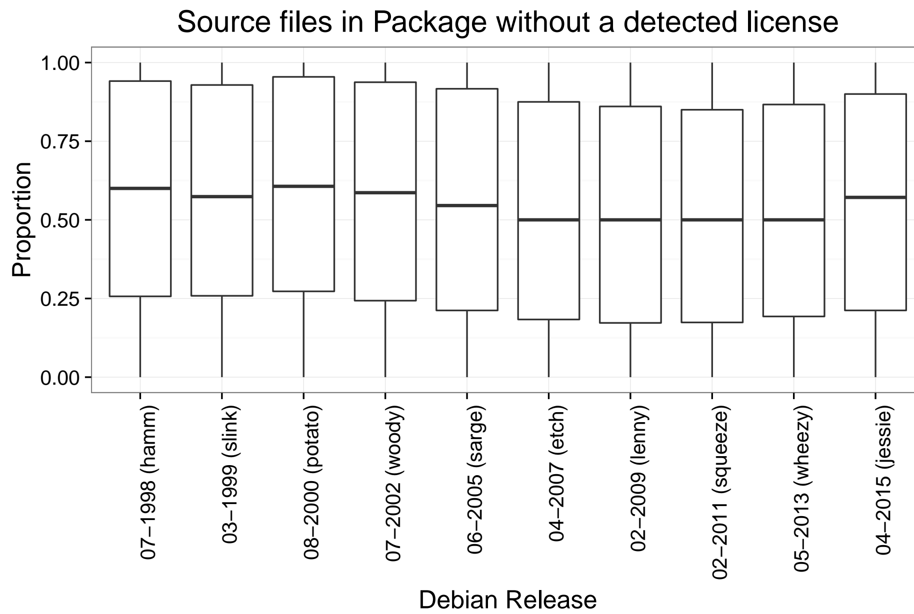


Figure 4.10: Box plots of the proportion of files with no identifiable license.

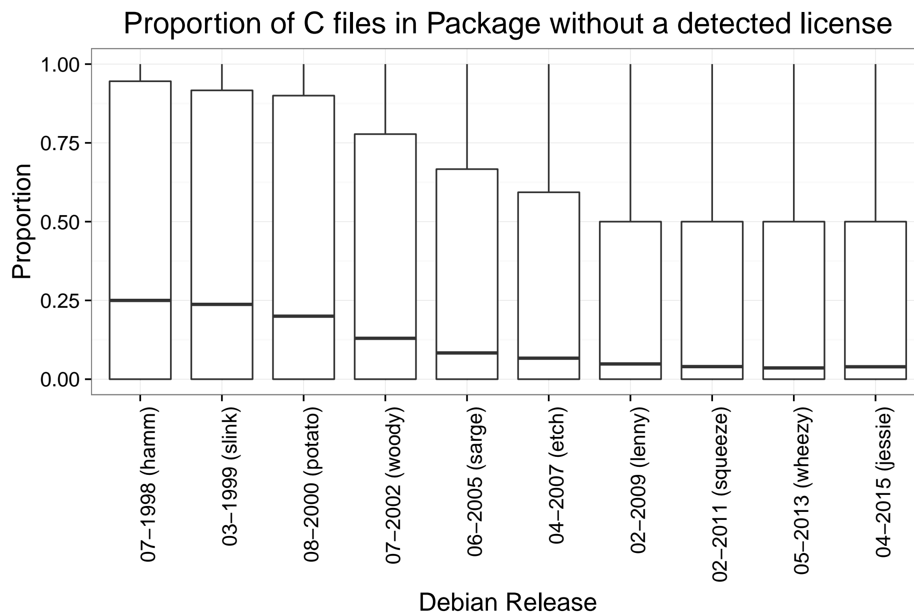


Figure 4.11: Box plots of the proportion of C files with no identifiable license.

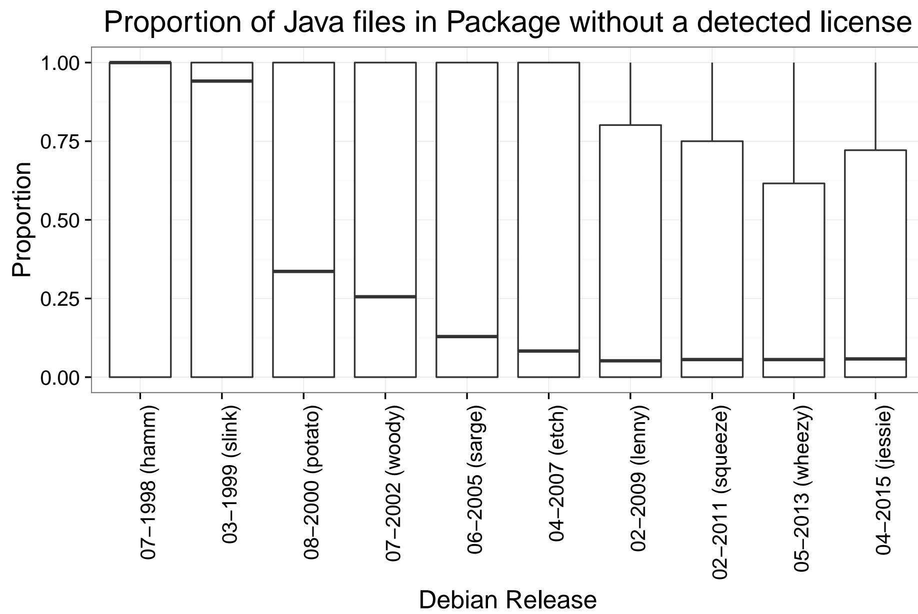


Figure 4.12: Box plots of the proportion of Java files with no identifiable license.

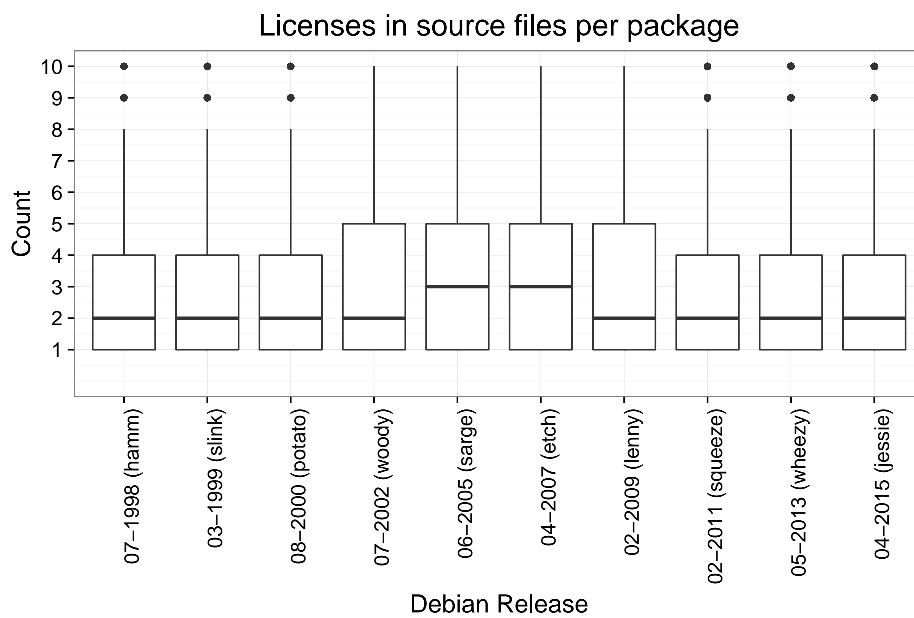


Figure 4.13: Box plots with the number of different identified licenses per package.

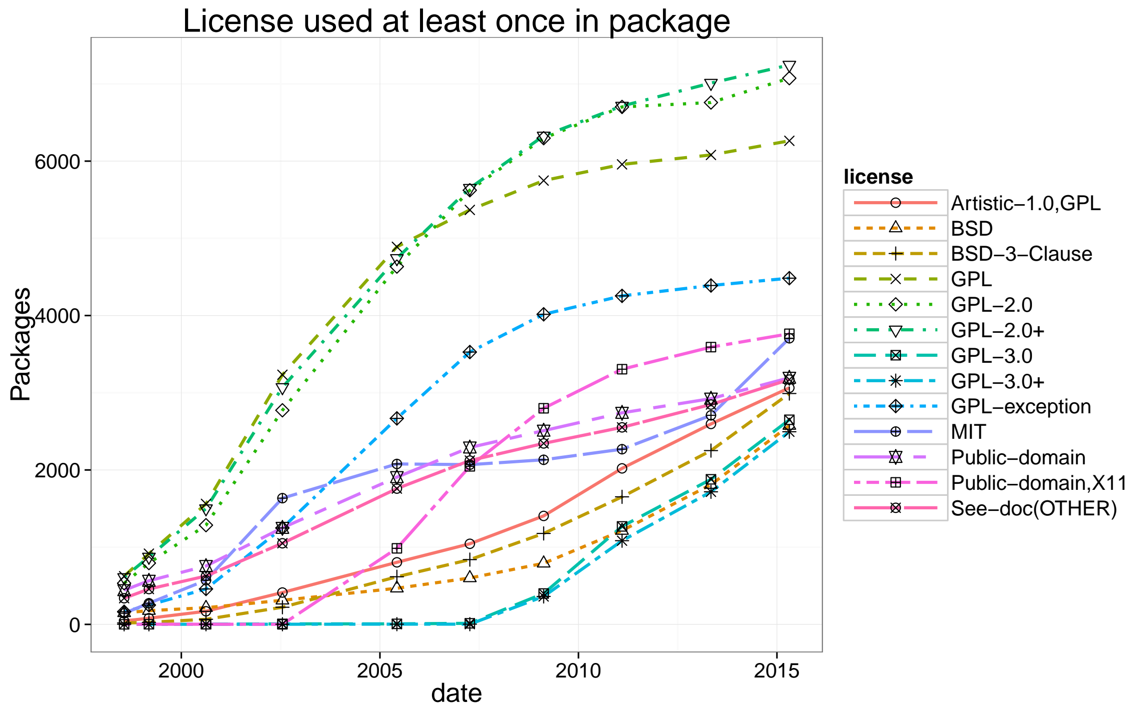


Figure 4.14: Evolution of the number packages that use a license at least once.

that occur at least once in a package. As it can be seen variants of the GPL licenses are still, by far, the most commonly used, and in particular versions 2.0 and 2.0+ (i.e., “version 2 or any later version”).

Figure 4.15 shows the evolution of dominant licenses in Debian packages, according to our definition. The top license is, once again, GPL-2.0+, followed by: Artistic-1.0/GPL dual-licensing (the licensing choice of Perl and most Perl libraries), GPL-3.0+, and Apache-2.0.

With regard to the variability of licenses in packages, we present in Figure 4.17 the box plot of the MNDif of the licenses per package in each release. As it can be seen, most packages have very small license variability, and license diversity seems to be decreasing over time. This might be due to new, popular programming language ecosystems that manage to impose, either with legal agreements or simply via “bandwagon” effects, a specific license to all the new modules and libraries that will be developed in the language.

Figure 4.16 shows a scatter plot showing MNDif vs the number of source files in a package, for the most recent Debian release (Jessie). A pattern stands out: the more source files in a package, the less license diversity. This might seem counter intuitive at first, because more files should give more opportunities for reusing code from other FOSS projects and hence adopt a new license, increasing diversity. Our intuition is that such aspect is countered by the fact that large, well-established FOSS projects tend to be governance-heavy, cautious when importing external code in their own repositories (e.g., due to long-term maintainability concerns), if not simply used to impose a specific

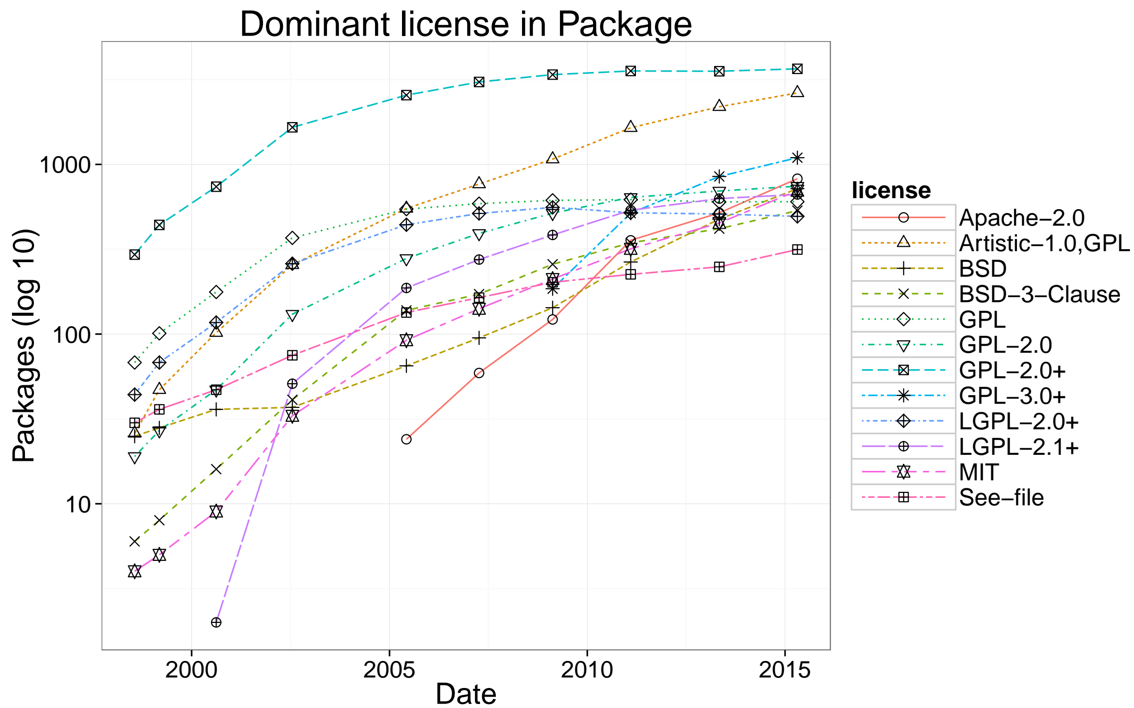


Figure 4.15: Evolution of the number of packages that have a given dominant license.

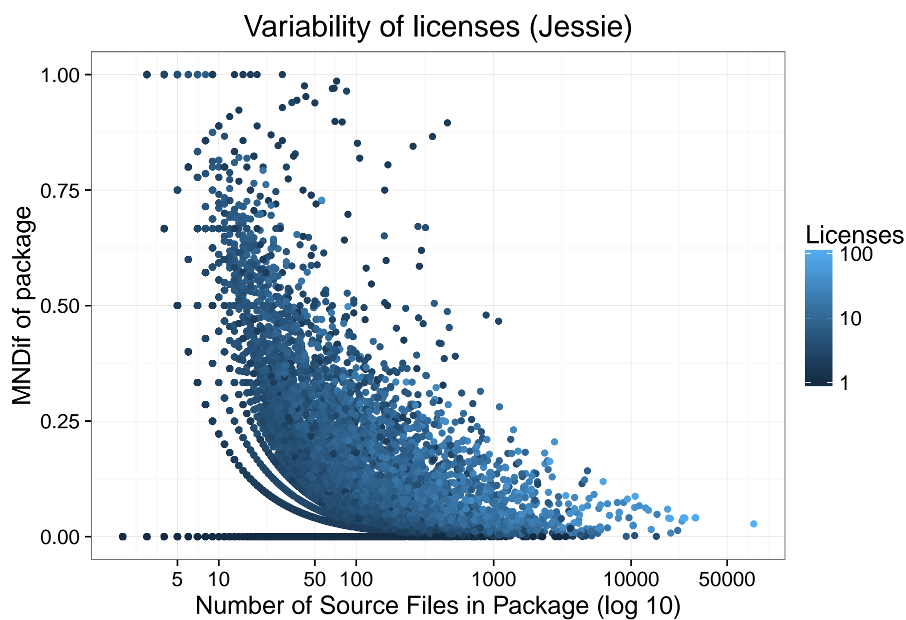


Figure 4.16: Each point represents a package in Jessie (version 8): its MNDif vs number of source files. As it can be seen, smaller packages tend to have more variability in their licensing.

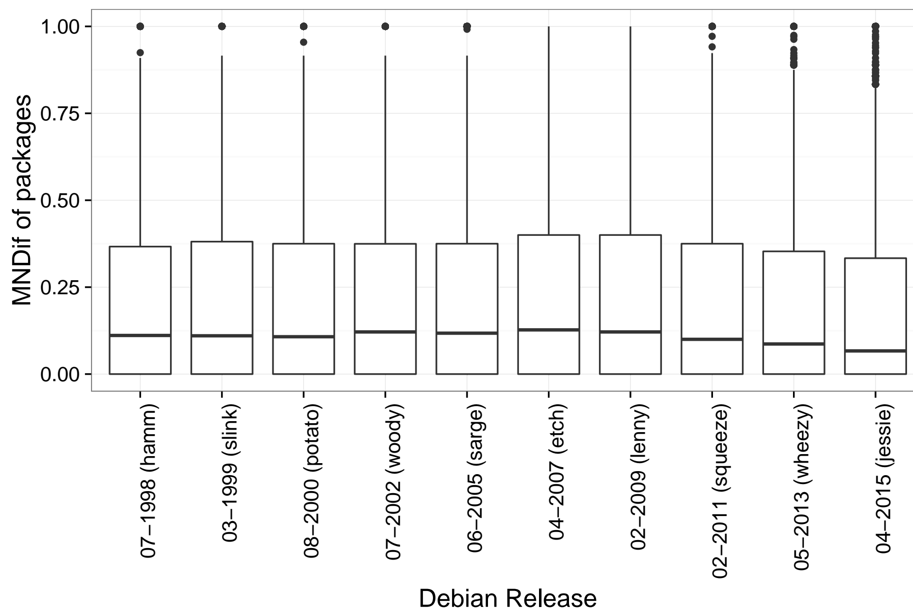


Figure 4.17: Box plots showing the MNDif of packages per release. A MNDif of zero means no variability, while 1 means that every license in the package is equally represented.

license as a requirement for accepting external code contributions.

To the best of our knowledge this study, originally presented in [30], is the first on FOSS license popularity at this scale, and in particular over such a long time frame. While there exist reports on the Web about FOSS license popularity, and most notably from Black Duck,[13] such reports do not disclose the adopted methodology nor are clear on the underlying sample of observed FOSS projects, making them non-reproducible. Furthermore they do not properly document how licenses are counted, which is an important and tricky aspect of surveying FOSS license use [99].

4.3.5 Validation

Looking back at this case study we can attempt a self-assessment of the advantages induced by using the Debsources Dataset as a starting point. Using Debsources Dataset metadata (the database) we have been able to study Debian growth over time as well as the correlation and distribution of the chosen metrics. We have also been able to get insights on software engineering practices such as package maintenance and study over time the popularity of programming languages and software licenses, aggregating at different granularities (file, package, release).

Without the Debsources Dataset the starting point of the case study would have necessarily been retrieving and unpacking all Debian releases, followed by running all measurement/mining tools on the obtained source code—incurring the costs of the needed storage and computation resources of course (documented in Section 4.2.2). None of this

[13] https://www.blackducksoftware.com/top-open-source-licenses

has been necessary to run our case study. More importantly than savings in computational resources, though, the Debsources Dataset relieves scholars from the responsibility of figuring out which-tools-to-use-when in order to mine Debian-specific data sources; the starting point becomes a relatively straightforward ER data model.

A counter argument here is that the metrics and information we were interested in were all already available in the dataset; unsurprisingly, given we initially included them in the dataset for our own needs. The first response to this is that a significant part of the metadata included in the dataset is intrinsic to either how Debian works (e.g., Debian release information) or the nature of the referenced objects (e.g., file size, checksums).

Second, when it comes to mining new facts that are not included in the Debsources Dataset, the source code part of the dataset and how it is organized offers many benefits. Most notably it saves space; thanks to deduplication the required disk space is cut by approximately 50%. As many kind batch source code analyses are I/O bound, an equivalent saving in processing time should generally be expected as well.

It is also reasonable to expect that newly mined facts from Debian source code will need to be correlated, one way or another, with metadata that *are* available in this dataset. The Debsources Dataset alleviates the need of having to mine them over and over again. It is at the border of source code mining and related metadata that the Debsources Dataset offers the most time- and space-saving opportunities for scholars of empirical software engineering, and FOSS in particular.

CHAPTER 5

Scaling to the entire software commons

This chapter is based on [57].

The Debsources experiment showcased some of the benefits of curating collections of FOSS artifacts and related metadata, so that researchers can work on them without having to go through the potentially error-prone heavy-lifting of assemblying them over and over again. According to several code metrics the Debsources Dataset can be considered "big", especially for FOSS software engineering standards. But if we put it in perspective of the entire corpus of FOSS the dataset is not that big; it "only" contains the entire history of Debian, no matter how large relevant the distribution is for distribution standards. As such, researchers can rely on the Debsources Dataset to draw *general* conclusions about FOSS only under the hypothesis that Debian is a representative sample of it. Such an hypothesis needs to be assessed on an experiment-by-experiment basis—sometimes it will be valid, others not.

Furthermore, Debsources is suitable for analyses of FOSS that stops at the macro-level [80], where the observable evolution points of software components are releases of individual components as shipped by curated collections like Debian. Sub-release evolution, e.g., changes performed during software development as captured by version control systems, are not observable at the Debsources level. In a sense, Debsources covers the spatial dimension of FOSS, stopping short of the entire space, and includes only a fairly coarse-grained view of its temporal dimension.

One can then wonder whether we can do any better in terms of coverage, while retaining the same good characteristics of Debsources—uniform organization, traceability, live updates with minimum lag, etc. Put it differently, how far can we go in terms of FOSS coverage while still preserving the good characteristics that would enable researchers to use such hypothetical source code archive as the basis for *any* empirical study targeting FOSS? Furthermore, can we at the same time increase significantly the granularity of the "snapshots" that are taken of the FOSS corpus?

Based on the work we have conducted over the past few years on the Software Heritage project [57],[1] we believe that we can go as comprehensive as it gets, that is, archiving

[1] https://www.softwareheritage.org

the entire body of FOSS, in source code form, while still retaining the good properties of Debsources and also capture every time-indexed, publicly available snapshot of its source code. More precisely we can archive the entire *software commons*—i.e., the whole body of software that is publicly available in source code form and that can be reused and modified with minimal restrictions. The software commons is a growing part of the broader information commons [100] which, of course, the raise of Free/Open Source Software (FOSS) has contributed enormously to nurture over the past decades [142].

In spite of its growing relevance for the industry, little action seem to have been put into large-scale archival and long-term preservation of FOSS source code. Comprehensive archives are available for a variety of digital objects, pictures, videos, music, texts, web pages, even binary executables [89]; but *source code* in its own merits has not yet been given the status of first class citizen in the digital archiving landscape.

In this final chapter we present the design and implementation of Software Heritage, an ambitious research initiative to collect, organize, preserve, and share the entire corpus of publicly accessible software source code. The project—co-founded by this thesis author and Roberto Di Cosmo—has been announced publicly for the first time in June 2016. Software Heritage is meant to cater for several use cases, one of which is large-scale empirical software engineering research on FOSS in the spirit of our research work as described in previous chapters. Other use cases—preservation of our cultural heritage as embedded in FOSS source code, industrial traceability of FOSS components, and curation of artifacts for computational education—are equally important and have been taken into accounts as design requirements.

5.1 Requirements

5.1.1 On the need of archiving FOSS source code

Despite the importance of FOSS in the development of science, industry, and society at large, it is easy to see that we are collectively not taking care of it properly. We briefly outline below some of the reasons why this is the case.

The FOSS development diaspora

With the meteoric rise of FOSS, millions of projects are now developed on publicly accessible code hosting platforms [144], such as GitHub, GitLab, SourceForge, Bitbucket, etc. Software projects also tend to move among those platforms over their lifetime, following current trends or the changing needs and habits of their developer community.

FOSS distribution channels are no less scattered. Some developers use code hosting platforms also for distribution. Other communities have their own archives organized by software ecosystems (e.g., CPAN, CRAN, ...). Then there are all FOSS distributions (Debian, Fedora, ...) and independent package management systems (npm, pip, OPAM, ...) which also retain copies of source code released elsewhere.

It is hence very difficult to appreciate the extent of the software commons as a whole: we direly need a single entry point—a modern "great library" of source code—where

one can find and study, for research purposes or otherwise, the evolution of all publicly available source code, independently of its development and distribution platforms.

The fragility of source code

We have known for a long time that digital information is fragile: human error, material failure, fire, hacking, can easily destroy valuable digital data, including source code. This is why carrying out regular backups is important. For users of code hosting platforms this problem may seem a distant one: the burden of "backups" is not theirs, but the platforms' one. As previously observed [162], though, most of these platforms are tools to enable collaboration and record changes, but do not offer any long term preservation guarantees: digital contents stored there can be altered or deleted over time.

Worse, the entire platform can go away, as we learned the hard way in 2015, when two very popular FOSS development platforms, Gitorious [76] and Google Code [87] announced shutdown. Over 1.5 million projects had to find a new accommodation since, in an extremely short time frame as regards Gitorious. Others have followed suit in subsequent months, including Microsoft's own forge CodePlex in 2017 [27]. This shows that the task of long term preservation cannot be assumed by entities that do not make it a stated priority: for a while, preservation may be a side effect of their missions, but in the long term it won't be.

We lack a comprehensive archive which undertakes this task, ensuring that if source code disappears from a given code hosting platform, or if the platform itself disappears altogether, the code will not be lost forever.

The very large telescope of source code, or lack thereof

With the growing importance of software, it is increasingly more important to provide the means to improve its quality, safety, and security properties. Sadly we lack a research instrument to analyze the whole body of publicly available source code.

To build such a "very large telescope" of source code—in the spirit of mutualized research infrastructures for physicists such as the Very Large Telescope [158] in the Atacama Desert or the Large Hadron Collider [85] in Geneva—we need a place where all information about software projects, their public source code, and their development history is made available in a uniform data model. This will allow to apply a large variety of "big code" techniques to analyze the entire corpus, independently of the origin of each source code artifact, and of the many different technologies currently used for hosting and distributing source code.

5.1.2 Goals

In order to address these three challenges, in June 2016 we unveiled the Software Heritage project, with initial support by Inria, with the stated goal to *collect, organize, preserve, and make easily accessible* all publicly available source code (a strict superset of all FOSS), independently of where and how it is being developed or distributed. The aim is to build a *common archival infrastructure*, supporting multiple use cases and applications

(see Section 5.1.4), but all exhibiting synergies with long-term safeguard against the risk of permanent loss of source code.

To give an idea of the complexity of the task, let's just review some of the challenges faced by the initial source code harvesting phase, ignoring for the moment the many others that arise in subsequent stages. First, we need to identify the code hosting places where source code can be found, ranging from a variety of well known development platforms to raw archives linked from obscure web pages. There is no universal catalog: we need to build one!

Then we need to discover and support the many different protocols used by code hosting platforms to list their contents, and maintain the archive up to date with the modifications made to projects hosted there. There is no standard, and while we hope to promote a set of best practices for preservation "hygiene", we must now cope with the current lack of uniformity.

We must then be able to crawl development histories as captured by a wide variety of version control systems [124]: Git, Mercurial, Subversion, Darcs, Bazaar, CVS, are just some examples of the tools that need to be supported. Also, there is no grand unifying data model for version control systems: one needs to be built.

To face such challenges, it is important that we computer scientist get directly involved: source code is the DNA of our discipline and we must be at the forefront when it comes to designing the infrastructure to preserve it in the very long term.

5.1.3 Core principles

Building the Software Heritage archive is a complex task, which requires long term commitment. To maximize the chances of success, we based this work on solid foundations, presented below as a set of core tenets for the project.

Transparency and FOSS

As stated by Rosenthal [134], in order to ensure long term preservation of any kind of information it is necessary to know the inner workings of all tools used to implement and run the archive. That is why Software Heritage develops and releases exclusively FOSS components to build its archive—from user-facing services down to the recipes of software configuration management tools used for the operations of project machines.

According to FOSS development best practices, the development of Software Heritage is conducted collaboratively on the project forge[2] and development communications happen on publicly accessible media (IRC channels, mailing lists, etc).[3]

Replication all the way down

There is a plethora of threats, ranging from technical failures to mere legal or even economic decisions, that might endanger long-term source code preservation. We know that

[2] https://forge.softwareheritage.org/
[3] https://www.softwareheritage.org/community/developers

they cannot be entirely avoided. Therefore, instead of attempting to create a system without errors, we design a system which tolerates them.

To this end, we will build replication and diversification in the system at all levels: a geographic network of mirrors, implemented using a variety of storage technologies, in various administrative domains, controlled by different institutions, and located in different jurisdictions. Releasing our own code as FOSS is synergistic with this goal, as it expected to further ease the deployment of mirrors by a variety of actors.

Multi-stakeholder and non-profit

Experience shows that a single for-profit entity, however powerful, does not provide sufficient durability guarantees in the long term. We believe that for Software Heritage it is essential to build a non-profit foundation that has as its explicit objective the collection, preservation, and sharing of the entire software commons.

In order to minimise the risk of having a single points of failure at the institutional level, this foundation needs to be supported by various partners from civil society, industry, and governments, and must provide value to all areas which may take advantage of the existence of the archive, ranging from the preservation of cultural heritage to research, from industry to education.

The foundation should be run transparently according to a well-documented governance and should be accountable to the public by reporting periodicly about its activities.

No a priori selection

A natural question that arises when building a long term archive is what should be archived in it among the many candidates available. In building Software Heritage we have decided to avoid any *a priori* selection of software projects, and rather archive them all. There are two main reasons underlying this design principle, one technical and one philosophical.

The first reason behind this choice is pragmatic: we have the technical ability to archive every software project available. Source code is usually small in comparison to other digital objects, information dense, and expensive to produce, unlike the millions of (cat) pictures and videos exchanged on social media. Additionally, source code is heavily redundant/duplicated, allowing for efficient storage approaches (see Section 5.2).

Second, thanks to FOSS software is nowadays massively developed in the open, so we get access to the history of software projects since their very early phase. This is a precious information for understanding how software is born and evolves and we want to preserve it for any “important” project. Unfortunately, when a project is in its infancy it is extremely hard to know whether it will grow into a king or a peasant. Consider PHP: when it was released in 1995 by Rasmus Lerdorf as PHP/FI (Personal Home Page tools, Forms Interpreter), who would have thought that it would have grown into the most popular Web programming language 20 years later?

Hence our approach to archive everything available: important projects will be pointed at by external authorities, emerging from the mass, less relevant ones will drift into oblivion.

Source code first

Ideally, one might want to archive software source code "in context", with as much information about its broader ecosystem: project websites, issues filed in bug tracking systems, mailing lists, wikis, design notes, as well as executable software binaries built for various platforms and the physical machines and network environment on which the software was run, allowing virtualization in the future. In practice, the resources needed for doing all this would be enormous, especially considering the *no a priori selection* principle, and we need to draw the scope line somewhere.

Software Heritage will archive *the entire source code* of software projects, together with their *full development history* as it is captured by state-of-the-art *version control systems* (or "VCS"). On one side, this choice allows to capture relevant context for future generations of developers—e.g., VCS history includes commit messages, precious information that detail *why* specific changes have been made to a given software at a given moment—and is precisely what currently nobody else comprehensively archives. Archiving VCS, as opposed to component releases only, is what accounts for the much finer grained granularity on the temporal dimension of Software Heritage in comparison to experiments like Debsources (Chapter 4).

On the other side, a number of other digital preservation initiatives are already addressing some of the other contextual aspects we have mentioned: the Internet Archive [89] is archiving project websites, wikis, and web-accessible issue trackers; Gmane [77] is archiving mailing lists; several initiatives aim at preserving software executables, like Olive [111], the Internet Archive, KEEP [66], E-ARK [64], and the PERSIST project [161], just to mention a few.

In a sense, Software Heritage embraces the Unix philosophy [132, 139] of doing one thing and doing it well, focusing on source code archival only, where its contribution is most relevant, and will go to great lengths to make sure that the source code artifacts it archives are easy to reference from other digital archives, using state-of-the-art linked data [24] technologies, paving the way to a future "semantic wikipedia" of software.

Intrinsic identifiers

The quest for the "right" identifier for digital objects has been raging for quite a while [18, 126, 162], and it has mainly focused on designing *digital identifiers* for objects that are not necessarily natively digital, like books or articles. Recent software development practices has brought to the limelight the need for *intrinsic identifiers* of natively digital objects, computed only on the basis of their *content* as a sequence of raw bytes.

Modern version control systems like Git [34] for example no longer rely on artificial opaque identifiers that need third party information to be related to the software artifacts they designate. They use identifiers that can be computed from, and verified on, the object itself and are tightly connected to it; we call these identifiers *intrinsic*. The SHA1 cryptographic hash [41] is the most used approach for computing them today. The clear advantage of crypto-hard intrinsic identifiers is that they allow to check that an obtained object is exactly the one that was requested without having to involve third party authorities.

Intrinsic identifiers also natively support integrity checks—e.g., you can detect alteration of a digital object for which an intrinsic identifiers was previously computed as a mismatch between its (current) content and its (previous) identifier—which is a very good property for any archival system.

Software Heritage will use intrinsic identifier for all archived source code. Pieces of information that are not natively digital, such as author or project names, metadata, or ontologies, non-intrinsic identifier will *also* be used. But for the long term preservation of the interconnected network of knowledge that is built natively by source code, intrinsic identifiers will be preferred.

Facts and provenance

Following best archival practices, Software Heritage will store full *provenance* information, in order to be able to always state *what* was found *where* and *when*.

In addition, in order to become a shared and trusted knowledge base, we push this principle further, and we will store only *qualified facts* about software. For example, we will not store bare metadata stating that the programming language of a given file is, say, `C++`, or that its license is `GPL3`. Instead we will store qualified statements saying that version `3.1` of the program `pygments` invoked with a given command line on this particular file reported it as written in `C++`; or that version `2.6` of the `FOSSology` license detection tool, ran with a given configuration (also stored), reported the file as being released under the terms of version 3 of the GPL license.

Minimalism

We recognize that the task that Software Heritage is undertaking is daunting and has wide ramifications. Hence we focus on building a core infrastructure whose objective is *only* collecting, organizing, preserving, and sharing source code, while establishing collaborations with any initiative that may add value on top or on the side of this infrastructure.

5.1.4 Use cases

A universal archive of software source code enables a wealth of applications in a variety of areas, broader than preservation for its own sake. Such applications are relevant to the success of the archive itself though, because long term preservation carries significant costs. Chances to meet them will be much higher if there are more use cases than just preservation, as the cost may then be shared among a broader public of potential archive users.

Cultural heritage

Software is everywhere: it powers industry and fuels innovation, it lies at the heart of the technology we use to communicate, entertain, trade, and exchange, and is becoming a key player in the formation of opinions and political powers. Software is also an essential mediator to access all digital information [33] and is a fundamental pillar of modern scientific research, across all fields and disciplines [163]. In a nutshell, software either

embodies a rapidly growing part of our cultural, scientific, and technical knowledge, or is a strict requirement to access it.

Looking more closely, though, it is easy to see that the actual *knowledge* embedded in software is not contained into executable binaries, which are designed to run on specific hardware and software platforms and that often become, once optimized, incomprehensible for human beings. Rather, knowledge is contained in software *source code* which is, as eloquently stated in the very well crafted definition found in the GPL license [78], "the preferred form [of a program] for making modifications to it [as a developer]".

As such, source code is starting to be recognized as a first class citizen in the area of cultural heritage, as a noble form of human production that needs to be preserved, studied, curated, and shared. Source code preservation is also an essential component of a strategy to defend against digital dark age scenarii [16] in which one might lose track of how to make sense of digital data created by software currently in production.

Software Heritage will be at the forefront of the preservation of the part of our cultural heritage that is embedded in software, as testified by the agreement[4] established between Inria and UNESCO on source code preservation, whose concrete actions will be carried on by Software Heritage.

Science

Big code research We have already discussed how the availability of a central repository where all the history of public software development is made available in a uniform data model will be a real asset for empirical software engineering and FOSS studies. It will enable unprecedented big data analysis both on the code itself and the software development process, unleashing a new potential for Mining Software Repository research [82].

Given its roots in the Debsources experience, even if its ambition was much more modest, serving this use case will be a key focus of Software Heritage.

Reproducibility in computer science In the long quest for making modern scientific results reproducible, and pass the scientific knowledge over to future generations of researchers, the three main pillars are: scientific articles, that describe the results, the data sets used or produced, and the software that embodies the logic of the data transformation, as shown in Figure 5.1.

Many initiatives have been taking care of two of these pillars, like OpenAire [121] for articles and Zenodo [169] for data, but for software source code, researchers keep pointing from their articles to disparate locations, if any, where their source code can be found: web pages, development forges, publication annexes, etc. And it shows, given the sad state of reproducibility in computer science when it comes to software artifacts, whose causes seem to lie for a large part in poor software traceability and availability practices [38, 74].

[4] http://fr.unesco.org/events/ceremonie-signature-du-partenariat-unescoinria-preservation-partage-du-patrimoine-logiciel

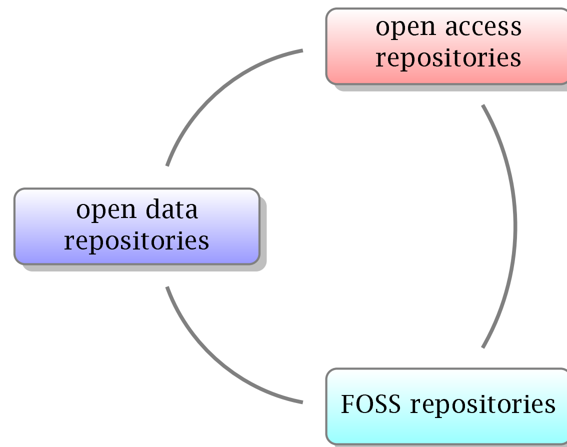


Figure 5.1: The scientific knowledge preservation trifecta

By providing a central archive for all publicly available source code, as well as a "deposit source code" service for scholars, Software Heritage will contribute a significant building block to the edifice of reproducibility in all fields of science.

Industry

Industry is growing more and more dependant of FOSS components, which are nowadays integrated in all kinds of products, for both technical and economic reasons. This tidal wave of change in IT brought new needs and challenges: ensuring technical compatibility among software components is no longer enough, one also needs to ensure compliance with several software licenses, as well as closely track software supply chain, and bills of materials to identify which specific variants of FOSS components were used in a given product.

Software Heritage makes two key contributions to the IT industry that can be leveraged in software processes. First, Software Heritage intrinsic identifiers can precisely pinpoint specific software versions, independently of the original vendor or intermediate distributor. This *de facto* provides the equivalent of "part numbers" for FOSS components that can be referenced in quality processes and verified for correctness independently from Software Heritage (precisely because they are intrinsic).

Second, Software Heritage will provide an open provenance knowledge base, keeping track of which software component—at various granularities: from project releases down to individual source files—has been found where on the Internet and when. Such a knowledge base can be referenced and augmented with other software-related facts, such as license and qualification information, and used by software build tools and processes to cope with current development challenges. Also, the previously discussed "deposit source code" service can also be leveraged by industries as a mechanism to publish complete and corresponding source code (CCSC) [78] bundles for the FOSS components they ship as part of their products, mutualizing the cost of maintaining such a service with other IT players.

The growing support and sponsoring for Software Heritage coming from industry

players like Microsoft, Huawei, Nokia, and Intel provides preliminary evidence that this potential is being understood.

Education

In library science a “sourcebook” is a collection of writings on a given topic that is often used in education as teaching support material for classes on that topic. In computer science education and, more generally, programming classes for the large public, the use of sourcebooks covering real source code is still very scarce in comparison to other disciplines.

Software Heritage can provide the solid basis on top of which organize curation of programming sourcebooks. Such an effort will allow to relate algorithms, data structure, and other programming techniques, often presented only in pseudo code for the sake of abstraction, to their real world implementations in a variety of languages. Doing so will allow students to see both faces—theory and practice—of what they are being taught.

Furthermore, it will be possible to track and follow the evolution of those implementations through time, going back several decades to when the studied techniques were first introduced, thanks to the native support of Software Heritage for software development history. While doing so, students will be able to appreciate metadata that add values to learning, such as developer comments as captured by VCS.

This of course won’t happen by itself. It will need collective work by motivated educators who can curate this work in the same style and spirit of the semantic wikipedia mentioned in Section 5.1.3. Still, once done, the results of this work will be permanent, as all corresponding software artifacts are, by construction, preserved in the Software Heritage archive.

5.2 Data model

In any archival project the choice of the underlying data model—at the *logical* level, independently from how data is actually stored on physical media—is paramount. The data model adopted by Software Heritage to represent the information that it collects is centered around the notion of *software artifact*, i.e., any piece of source code-related information that is extracted from software development and distribution platforms, is meaningful on its own right, and is addressable.

It is important to notice that, according to our principles, we must store with every software artifact full information about where it has been found. Therefore we start the description of our data model below by detailing the nature of provenance information as captured by Software Heritage.

5.2.1 Source code hosting places

Software Heritage relies on a curated list of *source code hosting places* to crawl as starting point to find software artifacts to archive.

The most common entries we expect to place in such a list are popular collaborative development forges (e.g., GitHub, Bitbucket), package manager repositories that host

source packages (e.g., CPAN, npm), and FOSS distributions (e.g., Fedora, FreeBSD). But we may of course allow also more niche entries, such as URLs of personal or institutional project collections not hosted on major forges.

While currently entirely manual, the curation of such a list might easily be semi-automatic, with entries suggested by fellow archivists and/or concerned users that want to notify Software Heritage of the need of archiving specific pieces of endangered source code. This approach is entirely compatible with Web-wide crawling approaches: Web crawlers capable of detecting the presence of source code might enrich the list. In both cases the list will remain curated, with (semi-automated) review processes that will need to pass before a hosting place starts to be used.

5.2.2 Software artifacts

Once the hosting places are known, they will need to be periodically looked at in order to add to the archive missing software artifacts. Which software artifacts will be found there?

In general, any software distribution mechanism will host multiple releases of a given software at any given time. For VCS this is the natural behaviour; for software packages, while a single version of a package is just *a* snapshot of the corresponding software product, one can often retrieve both current and past versions of the package from its distribution site.

By reviewing and generalizing existing VCS and source package formats, we have identified the following recurrent artifacts as commonly found at source code hosting places. They form the basic ingredients of the Software Heritage archive:[5]

file contents (AKA “blobs”) the raw content of (source code) files as a sequence of bytes, *without* file names or any other metadata. File contents are often recurrent, e.g., across different versions of the same software, different directories of the same project, or different projects all together.

directories a list of *named* directory entries, each of which pointing to other artifacts, usually file contents or sub-directories. Directory entries are also associated to arbitrary metadata, which vary with technologies, but usually includes permission bits, modification timestamps, etc.

revisions (AKA “commits”) software development within a specific project is essentially a time-indexed series of copies of a single “root” directory that contains the entire project source code. Software evolves when a developer modifies the content of one or more files in that directory and record their changes.

Each recorded copy of the root directory is known as a “revision”. It points to a fully-determined directory and is equipped with arbitrary metadata. Some of those are added manually by the developer (e.g., commit message), others are automatically synthesized (timestamps, preceding commit(s), etc).

releases (AKA “tags”) some revisions are more equals than others and get selected by developers as denoting important project milestones known as “releases”. Each

[5] as the terminology varies quite a bit from technology to technology, we provide both the canonical name used in Software Heritage and popular synonyms

release points to the last commit in project history corresponding to the release and might carry arbitrary metadata—e.g., release name and version, release message, cryptographic signatures, etc.

Additionally, the following crawling-related information are stored as provenance information in the Software Heritage archive:

origins code "hosting places" as previously described are usually large platforms that host several unrelated software projects. For software provenance purposes it is important to be more specific than that. Software origins are fine grained references to where source code artifacts archived by Software Heritage have been retrieved from. They take the form of ⟨*type*, *url*⟩ pairs, where *url* is a canonical URL (e.g., the address at which one can `git clone` a repository or `wget` a source tarball) and *type* the kind of software origin (e.g., `git`, `svn`, or `dsc` for Debian source packages).

projects as commonly intended are more abstract entities that precise software origins. Projects relate together several development resources, including websites, issue trackers, mailing lists, as well as software origins as intended by Software Heritage.

The debate around the most apt ontologies to capture project-related information for software hasn't settled yet, but the place projects will take in the Software Heritage archive is fairly clear. Projects are abstract entities, which will be arbitrarily nestable in a versioned project/sub-project hierarchy, and that can be associated to arbitrary metadata as well as origins where their source code can be found.

snapshots any kind of software origin offers multiple pointers to the "current" state of a development project. In the case of VCS this is reflected by *branches* (e.g., master, development, but also so called feature branches dedicated to extending the software in a specific direction); in the case of package distributions by notions such as *suites* that correspond to different maturity levels of individual packages (e.g., stable, development, etc.).

A "snapshot" of a given software origin records all entry points found there and where each of them was pointing at the time. For example, a snapshot object might track the commit where the master branch was pointing to at any given time, as well as the most recent release of a given package in the stable suite of a FOSS distribution.

visits links together software origins with snapshots. Every time an origin is consulted a new visit object is created, recording when (according to Software Heritage clock) the visit happened and the full snapshot of the state of the software origin at the time.

5.2.3 Data structure

With all the bits of what we want to archive in place, the next question is how to organize them, i.e., which logical data structure to adopt for their storage. A key observation for this decision is that source code artifacts are massively duplicated. That is so for several reasons:

- code hosting diaspora discussed in Section 5.1.1;

- copy/paste (AKA “vendoring”) of parts or entire external FOSS software components into other software products;
- large overlap between revisions of the same project: usually only a very small amount of files/directories are modified by a single commit;
- emergence of DVCS (*distributed* version control systems), which natively work by replicating entire repository copies around. GitHub-style pull requests are the pinnacle of this, as they result in creating an additional repository copy at each change done by a new developer;
- migration from one VCS to another—e.g., migrations from Subversion to Git, which are really popular these days—resulting in additional copies, but in a different distribution *format*, of the very same development histories.

These trends seem to be neither stopping nor slowing down, and it is reasonable to expect that they will be even *more* prominent in the future, due to the decreasing costs of storage and bandwidth.

For this reason we argue that any sustainable storage layout for archiving source code in the very long term should support *deduplication* at the granularity of individual software artifacts, allowing to pay for the cost of storing artifacts that are encountered more than once... only once. For storage efficiency, deduplication should be supported for all the software artifacts we have discussed, namely: file contents, directories, revisions, releases, snapshots.

Deduplication at an even finer granularity, e.g., sub-file for blobs, is possible and would increase storage efficiency even further. However it would also incur higher fragility risks due to the inherent additional complexity of reassembling software artifacts out of their fragments when needed.

Realizing the deduplication principle, the Software Heritage archive is conceptually a single (big) Merkle Direct Acyclic Graph [116] (DAG), as depicted in Figure 5.2. In such a graph each of the artifacts we have described—from file contents up to entire snapshots—corresponds to a node. Edges between nodes emerge naturally: directory entries point to other directories or file contents; revisions point to directories and previous revisions, releases point to revisions, snapshots point to revisions and releases. Additionally, each node contains all metadata that are specific to the node itself rather than to pointed nodes; e.g., commit messages, timestamps, or file names. Note that the structure is really a DAG, and not a tree, due to the fact that the line of revisions nodes might be forked and merged back.

In a Merkle structure each node is identified by an intrinsic identifier (as per our principles detailed in Section 5.1.3) computed as a cryptographic hash of the node content. In the case of Software Heritage identifiers are computed taking into the account both node-specific metadata and the identifiers of child nodes.

Consider the revision node shown in Figure 5.3. The node points to a directory, whose identifier starts with `fff3cc22...`, which has also been archived. That directory contains a full copy, at a specific point in time, of a software component—in the example a component that we have developed ourselves for the needs of Software Heritage. The revision node also points to the preceding revision node (`e4feb051...`) in the project

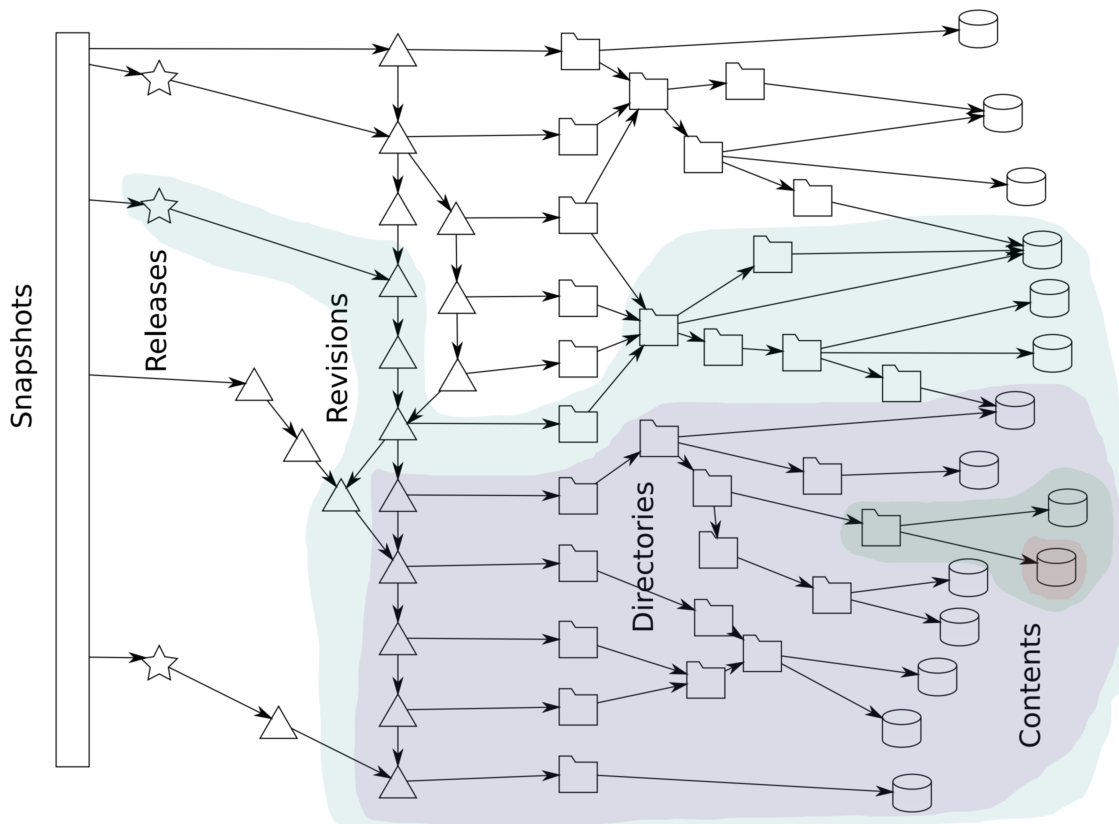


Figure 5.2: Software Heritage direct acyclic graph (DAG) data model

```
directory: fff3cc22cb40f71d26f736c082326e77de0b7692
parent: e4feb05112588741b4764739d6da756c357e1f37
author: Stefano Zacchiroli <zack@upsilon.cc>
date: 1443617461 +0200
committer: Stefano Zacchiroli <zack@upsilon.cc>
commiter_date: 1443617461 +0200
message:
  objstorage: fix tempfile race when adding objects

  Before this change, two workers adding the same
  object will end up racing to write <SHA1>.tmp.
  [...]
```

```
revision_id:  64a783216c1ec69dcb267449c0bbf5e54f7c4d6d
```

Figure 5.3: A revision node in the Software Heritage DAG

development history. Finally, the node contains revision-specific metadata, such as the author and committer of the given change, its timestamps, and the message entered by the author at commit time.

The identifier of the revision node itself (`64a78321...`) is computed as a cryptographic hash of a (canonical representation of) all the information shown in Figure 5.3. A change in any of them—metadata and/or pointed nodes—would result in an entirely different node identifier. All other types of nodes in the Software Heritage archive behave similarly.

The Software Heritage archive inherits useful properties from the underlying Merkle structure. In particular, deduplication is built-in. Any software artifacts encountered in the wild gets added to the archive only if a corresponding node with a matching intrinsic identifier is not already available in the graph—file content, commits, entire directories or project snapshots are all deduplicated incurring storage costs only once.

Furthermore, as a side effect of this data model choice, the entire development history of all the source code archived in Software Heritage—which ambitions to match all published source code in the world—is available as a unified whole, making emergent structures such as code reuse across different projects or software origins, readily available. Further reinforcing the use cases described in Section 5.1.4, this object could become a veritable "map of the stars" of our entire software commons.

5.3 Architecture

Both the data model described in the previous section and a software architecture suitable for ingesting source code artifacts into it have been implemented as part of Software Heritage.

5.3.1 Listing

The ingestion data flow of Software Heritage is shown in Figure 5.4. Ingestion acts like most search engines, periodically crawling a set of "leads" (in our case the curated list of code hosting places discussed in Section 5.2) for content to archive and further leads. To facilitate software extensibility and collaboration, ingestion is split in two conceptual phases though: listing and loading.

Listing takes as input a single hosting place (e.g., GitHub, PyPi, or Debian) and is in charge of enumerating all software origins (individual Git or Subversion repositories, individual package names, etc.) found there at listing time.

The details of how to implement listing vary across hosting platforms, and dedicated lister software components need to be implemented for each different *type* of platform. This means that dedicated listers exist for GitHub or Bitbucket, but that the GitLab lister—GitLab being a platform that can be installed on premises by multiple code hosting providers—can be reused to list the content of any GitLab instance out there.

Listing can be done fully, i.e., collecting the entire list of origins available at a given hosting place at once, or incrementally, listing only the new origins since the last listing. Both listing disciplines are necessary: full listing is needed to be sure that no origin is

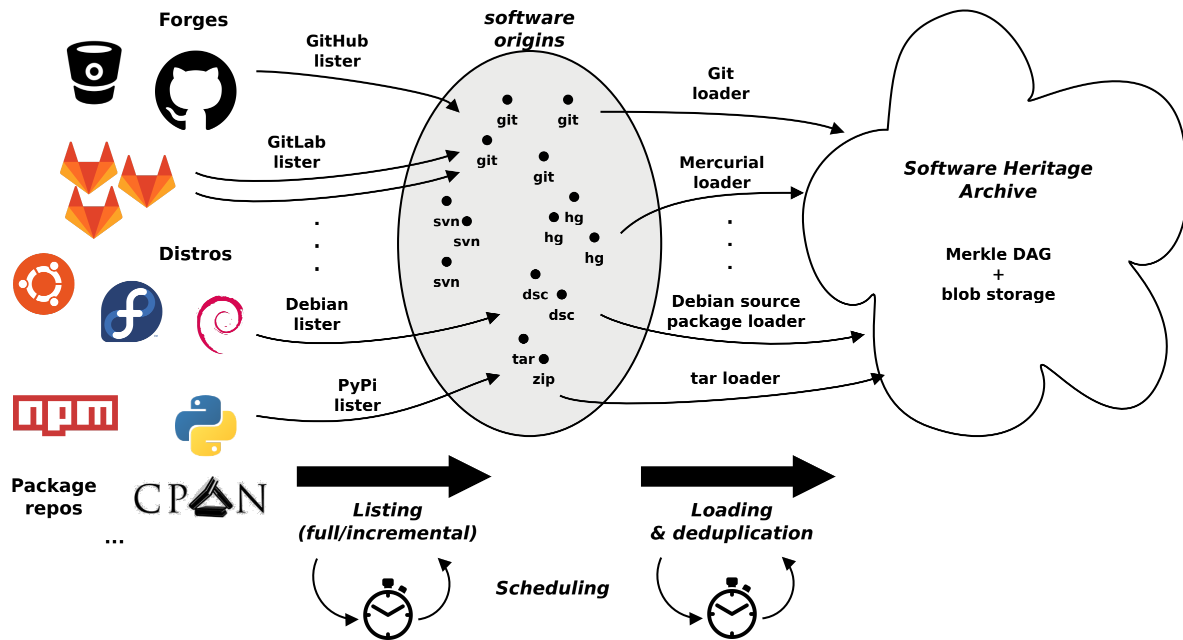


Figure 5.4: Ingestion data flow in Software Heritage

being overlooked, but it might be unwieldy if done too frequently on large platforms (e.g., GitHub, with more than 55 million Git repositories as of early 2017), hence the need of incremental listing to quickly update the list of origins available at those places.

Also, listing can be performed in either pull or push style. In the former case the archive periodically checks the hosting places to list origins. In the latter code hosting sites, when properly configured to work with Software Heritage, contact back the archive at each change in the list of origins. Push looks appealing at first and might minimize the lag between the appearance of a new software origin and its ingestion in Software Heritage. On the other hand push-only listing is prone to the risk of losing notifications that will result in software origins not being considered for archival. For this reason we consider push an optimization to be added on top of pull in order to reduce lag where applicable, rather than an on par alternative to pull.

5.3.2 Loading

Loading is the process responsible of actually ingesting into the archive the source code found at known software origins.

Loaders are the software components in charge of extracting source code artifacts from software origins and adding them to the archive. Loaders are specific to the technology used to distribute source code: there will be one loader for each type of version control system (Git, Subversion, Mercurial, etc.) as well as one for each source package format (Debian source packages, source RPMs, tarballs, etc).

Loaders natively deduplicate with respect to the entire archive, meaning that any artifact (file content, revision, etc.) encountered at any origin will be added to the archive only if a corresponding node cannot be found in the archive as a whole.

Consider the Git repository used for the development of the Linux kernel, which is fairly big, totaling 2 GB on disks for more than 600 000 revisions and also widely popular with thousands of (slightly different) copies available only on GitHub. At its first encounter ever, the Git loader will load essentially all its file contents, revisions, etc., into the Software Heritage archive. At the next encounter of an identical repository, nothing will be added at all. At the encounter of a slightly different copy, e.g., a repository containing a dozen additional commits not yet integrated in mainline Linux, only the corresponding revision nodes, as well as the new file contents and directories pointed by them, will be loaded into the archive.

5.3.3 Scheduling

Both listing and loading happen periodically on a schedule.[6] The *scheduler* component of Software Heritage is in charge of keeping track of when the next listing/loading actions need to happen, for each code hosting place (for listers) or software origins (for loaders).

While the amount of hosting places to list is not enormous, the amount of individual software origins can easily reach the hundreds of millions given the current size of major code hosting places. Listing/loading from that many Internet sites too frequently would be unwise in terms of resource consumption, and also very likely unwelcome by the maintainers of those sites. This is why we have adopted an adaptive scheduling discipline that strikes a good balance between update lag and resource consumption.

Each run of periodic action, such as listing or loading, can be “fruitful” or not. It is fruitful if and only if it resulted in new information, with respect to the last visit, being added to the archive. For instance, listing is fruitful when it results in the discovery of new software origins; loading is if the overall state of the consulted origin differs from the last observed one.

If a scheduled action has been fruitful, it means that the consulted site has seen at least some activity since the last visit, and we will increase the frequency at which that site will be visited in the future. In the converse case (no activity), visit frequency will be decreased.

Specifically, Software Heritage adopts an *exponential backoff* strategy, in which the visit period is halved when activity is noticed, and doubled when no activity has been observed. Currently, the fastest a given site will be consulted is twice day (i.e., every 12 hours) and the slowest is every 64 days. Early experiences with large code hosting sites such as GitHub seem to tell that ≈90% of the repositories hosted there quickly fall to the slowest update frequency (i.e., they don’t see any activity in 2-month time windows), with only the remaining ≈10% seeing more activity than that.

5.3.4 Archive

At a logical level, the Software Heritage archive corresponds to the Merkle DAG data structure described in Section 5.2. On disk, the archive is stored using different tech-

[6] as discussed, even when listing is performed in push style, we still want to periodically list pull-style to stay on the safe side, so scheduling is always needed for listing as well

nologies due to the differences in the size requirements for storing different parts of the graph.

File content nodes require the most storage space as they contain the full content of all archived source code files. They are hence stored in a key-value object storage that uses as keys the intrinsic node identifiers of the Merkle DAG. This allows trivial distribution of the object storage over multiple machines (horizontal scaling) for both performance and redundancy purposes. Also, the key-value access paradigm is very popular among current storage technologies, allowing to easily host additional copies of the archive either on premise or on public cloud offerings.

The rest of the graph is stored in a relational database (RDBMS), with roughly one table per type of node. Each table uses as primary key the intrinsic node identifier and can easily be sharded (again, with horizontal scaling) across multiple servers. Master/slave replication and point-in-time recovery can be used for increased performance and recovery guarantees. There is no profound reason for storing this part of the archive in a RDBMS, but for what is worth our early experiments seem to show that graph database technologies are not yet up to par with the size and kind of graph that Software Heritage already is with its current coverage (see Section 5.4).

A weakness of deduplication is that it is prone to hash collisions: if two *different* objects hash to the same identifier there is a risk of storing only one of them while believing to have stored them both. For this reason, where checksums algorithms are no longer considered strong enough for cryptographic purposes,[7] we use multiple checksums, with unicity constraints on *each* of them, to detect collisions before adding a new artifact to the Software Heritage archive. For instance, we do not trust SHA1 checksums alone when adding new file contents to the archive, but also compute SHA256, and “salted” SHA1 checksums (in the style of what Git does). Also, we are in the process of adding BLAKE2 checksums to the mix.

Regarding mirroring, each type of node is associated to a change feed that takes note of all changes performed to the set of those objects in the archive. Conceptually, the archive is append-only, so under normal circumstances each feed will only lists additions of new objects as soon as they get ingested into the archive. Feeds are persistent and the ideal branching point for mirror operators who, after an initial full mirror step, can cheaply remain up to date with respect to the main archive.

On top of the object storage, an archiver software component is in charge of both enforcing retention policies and automatically heal object corruption if it ever arises, e.g., due to storage media decay. The archiver keeps track of how many copies of a given file content exist and where each of them is. The archiver is aware of the desired retention policy, e.g., “each file content must exist in at least 3 copies”, and periodically swipe all known objects for adherence to the policy. When fewer copies than desired are known to exist, the archiver asynchronously makes as many additional copies as needed to satisfy the retention policy.

The archiver also periodically checks each copy of all known objects—randomly selecting them at a suitable frequency—and verifies it for integrity by recomputing its intrinsic identifier and comparing it with the known one. In case of mismatch all known

[7] note that this is already a higher bar than being strong enough for archival purposes

copies of the object are checked on-the-fly again; assuming at least one pristine copy is found, it will be used to overwrite corrupted copies, “healing” them automatically.

5.4 Current status and roadmap

The Software Heritage archive grows incrementally over time as long as new listers/loaders get implemented and periodically run to ingest new content.

5.4.1 Listers

In terms of listers, we initially focused on targeting GitHub as it is today by far the largest and most popular code hosting platform. We have hence implemented and put in production a GitHub lister, capable of both full and incremental listing. Additionally, we have recently put in production a similar lister for Bitbucket. Common code among the two has been factored out to an internal lister helper component that can be used to easily implement listers for other code hosting platforms.[8] Upcoming listers include FusionForge, Debian and Debian-based distributions, as well as a lister for bare bone FTP sites distributing tarballs.

5.4.2 Loaders

Regarding loaders, we initially focused on Git as, again, the most popular VCS today. We have additionally implemented loaders for Subversion, tarballs, and Debian source packages. A Mercurial loader is in the working.

5.4.3 Archive coverage

Using the above software components we have already been able to assemble what, to the best of our knowledge, is the largest software source code archive in existence.

We have fully archived once, and routinely maintain up-to-date since, GitHub into Software Heritage, for more than 50 million Git repositories. GitHub itself has acknowledged Software Heritage role as 3rd-party archive of source code hosted there.[9]

Additionally we have archived, as one shot but significant in size archival experiments, all releases of each Debian package in between 2005–2015, and all current and historical releases of GNU projects as of August 2015. We have also retrieved full copies of all repositories that were previously available from Gitorious and Google Code, now both gone. All Git repositories previously available on those forges have been injected into Software Heritage; loading of the Subversion and Mercurial repositories previously available there is in our backlog.

In terms of storage, each copy of the Software Heritage object storage currently occupies $\approx$150 TB of individually compressed file contents. The average compression ration is 2x, corresponding to 300 TB of raw source code content. Each copy of the RDBMS used to store the rest of the graph (Postgres) takes $\approx$5 TB. We currently maintain 3 copies of

[8] see https://www.softwareheritage.org/2017/03/24/list-the-content-of-your-favorite-forge-in-just-a-few-steps/ for a detailed technical description

[9] https://help.github.com/articles/about-archiving-content-and-data-on-github/

the object storage and 2 copies of the database, the latter with point-in-time recovery over a 2-week time window.

As a logical graph, the Software Heritage Merkle DAG has ≈5 billion nodes and ≈50 billion edges. We note that more than half of the nodes are (unique) file contents (≈3 B) and that there are ≈750 M revision/commit nodes, collected from ≈55 M origins.

5.4.4 Features

The following functionalities are currently available for interacting with the Software Heritage archive:

content lookup allows to check whether specific file contents have been archived by Software Heritage or not. Lookup is possible by either uploading the relevant files or by entering their checksum, directly from the Software Heritage homepage.

browsing via API allows developers to navigate through the entire Software Heritage archive as a graph. The API offered to that end is Web-based and permits to lookup individual nodes (revisions, releases, directories, etc.), access all their metadata, follow links to other nodes, and download individual file contents. The API also gives access to visit information, reporting when a given software origin has been visited and what its status was at the time.

The API technical documentation[10] has many concrete examples of how to use it in practice.

The following features are part of the project technical road map and will be rolled out incrementally in the near future:

Web browsing equivalent to API browsing, but more convenient for non-developer Web users. The intended user interface will resemble state-of-the art interfaces for browsing the content of individual version control systems, but will be tailored to navigate a much larger archive.

provenance information will offer "reverse lookups" of sort, answering questions such as "give me all the places and timestamps where you have found a given source code artifact". This is the key ingredient to address some of the industrial use cases discussed in Section 5.1.4.

metadata search will allow to perform searches based on project-level metadata, from simple information (e.g., project name or hosting place), to more substantial ones like the entity behind the project, its license, etc.

content search conversely, content search will allow to search based on the *content* of archived files. Full-text search is the classic example of this, but in the context of Software Heritage content search can be implemented at various level of "understanding" of the content of individual files, from raw character sequences to full-fledged abstract syntax trees for a given programming language.

[10] https://archive.softwareheritage.org/api/

Chapter 6

Conclusion and future work

We conclude this research overview by briefly recapitulating our research work thus far and sketching a few research directions for the future.

Our research journey over the past decade followed an organic evolution path through the analysis of component-based FOSS systems. It started by investigating the state-of-the-art of package life-cycle management in FOSS distributions and observed that, at the time, many issues were still plaguing upgrades [54]. Our contributions in the context of the Mancoosi project (Chapter 2) have all been geared towards addressing those issues.

Quality assurance for FOSS distributions On the distribution-editor side we have built upon seminal work by the EDOS project [114] and introduced the notion of *strong dependency* [1] as a mechanism to efficiently pinpoint critical packages that should be handled with care in large repositories, and contributed to the development of the notion of *repository futures* [7, 9] that serves the related need of identifying packages that need manual intervention to become installable again. We have had the chance to look back at all the work done—by ourselves and a few other groups around the world—on the front of formal analyses of package dependencies in FOSS collections [153, 53]. We found that it has had a significant, positive impact on the quality of curated FOSS collections, most notably on the amount of non-installable packages that get distributed to users over time [3].

While that line of work focused on the static aspects of packages, such as dependencies and conflicts, we have also worked on the dynamic aspects that might adversely impact software upgrades. In particular, we have worked on the classification and simulation of *maintainer scripts*, using meta-modeling techniques [59, 35, 45]. In between static and dynamic package features, we have studied the origin of *inter-package conflicts* [19, 20], which are visible at the abstract level of packages, but generally stem from lower-level aspects like the layout of installed files on disk or indeed the behavior of maintainer scripts.

Outside the context of the Mancoosi project, and not detailed in this manuscript, we have contributed to a different aspect of distribution quality assurance: data warehousing for FOSS metadata. We have developed the Ultimate Debian Database [119] (UDD) to consolidate Debian and Ubuntu distribution metadata to server a number of use cases, from quality assurance (for distribution maintainers) to data mining (for researchers).

UDD has been in production in Debian/Ubuntu since then, it has been used as a basis for the well-known MSR mining challenge competition [86] in 2010, and it tied thematically well into subsequent work of ours on Debsources and Software Heritage.

Better dependency solving for FOSS distributions On the distribution-user side we have worked to improve the quality of *dependency solving* by inducing a synergy between package manager engineers (who needed to realize that *ad hoc* solving solutions were showing their limits, like incompleteness and poor request expressivity) and constraint solving researchers (who are generally eager to tackle challenging problems coming from the real world).

This line of work resulted in standardized formats and languages, like *CUDF* [154, 156], equipped with formal semantics that can be used as a *lingua franca* for dependency solving among package managers and constraint solvers [4]. We have then used CUDF—and its the reference implementation that we authored: *libCUDF*—to affirm and establish better engineering practices and design patterns for package managers [5, 8]. To further adoption and impact of these principles we have run the MISC *dependency solving competition* [6], which resulted in FOSS dependency solvers that are nowadays used in production, either as plug-in solvers for major distributions (this is the case for Debian) or as the main building block for novel package managers (this is the case of OPAM [4]) that have adopted since day 1 the software architecture we introduced.

We have also worked on a side topic not detailed in this manuscript: bridging the gap between nearby but previously unrelated formalisms for modeling software components. In particular, we have introduced a one-way mapping from *software product lines* (SPL) expressed as feature diagrams to distribution packages [56]. This has paved the way to reusing package research results and technologies in large industrial settings, where SPL have seen large adoption.

Component modeling for the "cloud" To expend both modeling and automated deployment of FOSS components outside the boundaries of individual machines we have introduced a *component model for cloud- and networked-applications*, called Aeolus [58] (see Chapter 3). It allows to capture intra-machine package details like dependencies and conflicts, but also inter-machine aspects such as running services and their requirements, capacity constraints, and the dynamic spawning and disposal of cloud resources.

As the expressivity of the Aeolus component model implies that operations that we can take for granted for packages are either non decidable or computationally very hard, Aeolus and its variants underwent thorough explorations of what can be automated on top of them, which resulted in several theoretical publications on the topic [51, 52, 48].

To overcome some of the proven limitations when working with this expressivity we have worked on automating deployment of cloud and networked applications, creating pragmatic FOSS planning and deployment tools, based on more limited variants of the full Aeolus component model, that have seen relevant industrial adoption such as *Zephyrus* [32, 50, 46] and its companion tools in the Aeolus suite.

FOSS source code analysis in the large As the most recent step of our research journey thus far, we have turned our attention to delving into large collections of FOSS *source*

code. It was natural for us to do so because we felt we had reached the limits of the quality improvements that could be delivered by stopping at the abstraction level of inter-package relationships. It was time to look at the roots of those relationships, that is, the actual code implementing packages of which dependencies and conflicts provide a coarse-grained abstract view.

Out of first-hand research need, we took what seemed at the time only a quick detour to design and implement a generic platform that would allow to perform large-scale analyses on the source code of Debian: Debsources (see Chapter 4) was born [31]. In addition to the platform itself, that has since been adopted by Debian in production and still is today, Debsources has been used to polish and publish one of the largest open dataset in existence about long-term FOSS evolution: the Debsources Dataset [168, 30].

Software Heritage (Chapter 5) came to be as the conjunction point of several interests. On the one hand, it is the natural expansion of Debsources to its largest extent possible: collecting, organizing, and sharing *all publicly available source code* with all researchers out there so that they can systematize analyses of our entire Software Commons. But on the other hand Software Heritage is also much more, due to synergies that become visible only at its scale and that indeed were not apparent in the more limited scope of Debsources. *FOSS preservation* became a natural concern that Software Heritage can address, given its comprehensiveness; it is in fact a concern that the project *should* address, given the task is currently unattended to in the digital preservation landscape. Other use cases became apparent as well, such as easing the tracking of FOSS for industrial needs, helping out with the reproducibility of science when it comes to the use of FOSS as part of scientific experiments, and helping educators with the assembly of the ultimate source book for computing education.

To tackle all these issues we have prototyped, designed, and launched Software Heritage [57], that has already become the largest source code archive in existence and credited as the premier project for source code preservation in the world by renowned scientific, industrial, and institutional partners.

6.1 Research directions

Software Heritage is still far from realizing its full mission though. Aside from technical work—which are "Simple Matter[s] of Programming", as the software engineering expression goes—many scientific challenges will need to be tackled for the project to succeed. Some of them are summarized below.

Data model and storage The chosen data model for Software Heritage (a Merkle DAG) is the best fit for the archival of software artifacts that are massively redundant over the Internet, and increasingly more so, and hence need to be subject to *de*duplication for their long-term archival to be viable. But such a model has its own drawbacks.

Most notably, to answer provenance queries of the form *"tell me all the places and times where/when you have encountered this source code artifact"* (e.g., a specific file content or commit) one needs to solve a single-source shortest path problem on the entire graph, whose size is, at the time of writing, approximately 7 billion nodes and 70

billion edges. While algorithmically the complexity of the problem is linear in the size of the graph, I/O costs makes the problem practically unwieldy when the graph does not fit in primary memory (RAM).

Research techniques to deal with even larger graphs on secondary storage exist [113, 136, 135], but they do not work well on the Software Heritage graph which does not appear to exhibit small world properties, and hence is not amenable to efficient treatment with distributed approaches that require computations to quickly complete after a few iterative rounds. General purpose state-of-the-art graph database products do not seem to fare better and generally rule out the possibility of dealing with graphs of the Software Heritage size.

While it looks possible (at least for now...) to make at least the naked graph structure fit into main memory and work from there, the challenge of finding the most appropriate data model, or physical representation of it, for storing the Software Heritage canonical graph in a way that is amenable to cheap and scale-out processing is up.

Metadata alignment In the cases of Debsources and UDD metadata management didn't pose much of a problem: well-defined metadata schemata existed in the origin context that the two infrastructure were meant to represent, so we just needed to store the corresponding metadata and use them as needed. In the context of Software Heritage things get more complicated, because the archive contains source code artifacts coming from very different projects that might use very different, and often inconsistent, schemata to encode their metadata. Even worse, the same artifacts Software Heritage archives can be found in different locations that might use different metadata schemata and/or provide contradicting metadata within the same schema.

The underlying problem here is schema matching [130], which is well-known in data warehousing and frequently emerges in corporate merger situations. However, Software Heritage is faced with the extra complexity of wanting to both preserve the original software metadata as closely as possible, together with accurate provenance information as we do for other archived information, *and* to abstract over all found metadata—no matter how inconsistent and heterogeneous they might be—to provide some basic metadata-based search functionalities that should have chances to find results across the archive.

The number of different software ontologies out there is very significant and appears to be growing—Schema.org, FOAF, DOAP, etc. are just some of the most used ontologies and technologies in the domain [81, 28, 60].[1] Software Heritage will need to deal with this heterogeneity and resulting inconsistencies, striking a good balance between faithful preservation and practical functionalities for its users.

Software Heritage will also be the natural playground where to synthesize *missing* metadata in any given anthology applying, for instance, supervised machine learning approaches that could then leverage *available* metadata as training/learning sets.

Software phylogenetics As we have seen, public software development routinely migrate from one platform to another. Furthermore, due to their inherent licensing properties, FOSS software components and even individual code snippets get embedded, with

[1] see CodeMeta [94] for a recent attempt at surveying most of them

or without modifications, into projects other than those where they originated from. As of today Software Heritage seems to be the only dataset where one could observe to its largest extent possible the *impact* that any given line of publicly released source code has had in terms of adoption, adaptation, and reuse. This is so due to: *a)* the comprehensiveness of the archive, and *b)* the data model that deduplicates, and hence keeps closely together, all source code artifacts encountered in the wild.

To reap the benefits of this potential we will need to make clone detection techniques [137, 97] work at this scale, as they currently consider to be "large scale" collections of software development projects that are one order of magnitude smaller than the Software Heritage [138]. Research communities that are interested in this topic abound [103, 128], but do not seem to have had access to such a vast playground as of yet. We will need to work with them on the preliminary required step of defining the most appropriate access mechanisms and scientific APIs to exploit the archive.

Code search at scale Code search, one of the features on the Software Heritage roadmap, is an open research challenge as well. Generally speaking, code search can be implemented at very different level of "understanding" of source code.

At the richest level, any (syntactically correct) source code file can be parsed into an abstract syntax tree (AST) and indexed as such using a number of proven AST indexing techniques and technologies. But even letting aside for a moment the scale issue, the Software Heritage archive is very heterogeneous in terms of programming languages and also of their *versions*, which have evolved significantly and often in non backward compatible ways over the time frame of source code that is already archived in Software Heritage today. Finding the right parser for any given file stored in the archive, especially considering that it might appear with many different names (something that programming languages detectors tend not to like), is already a challenging problem *per se*. One could then either resort to partial indexing, allowing to search only a limited set of programming languages that can both be recognized as such and parsed, as it as already been done at a much smaller scale [63], or resort to indexing approaches that "understand" much less of the actual structure of source code.

At the extreme end of low-understanding approaches we can find reverse indexes approach with very simple stemming: just split the content of source code files into "words", using any sequence of non-alphanumeric character as word separators. This works at much larger scales than Software Heritage (like the entire Web, which is routinely indexed by search engines), but offers poor expressivity to the developers that will actually perform the searches.

A sweet spot in the spectrum, that have been successfully applied in the context of Debsources, is using trigram-based reverse indexes [146, 170]. Doing so allows to support regular expression based searches on large code bases [39], without having to actually "grep" through the actual content of the files at search time, which is daunting at large scale due to I/O costs. However, while this has proven to be viable on relatively cheap hardware for the code shipped by the development release of Debian [146] (with more than 1 billion source lines of code), Software Heritage is projected to be 2–3 orders of magnitude larger, meaning that indexes can no longer be made to fit in primary memory without incurring non-sustainable hardware costs. Different, very likely

distributed/scale-out approaches will need to be found to make the approach viable at the Software Heritage scale.

Working on some of these challenges will require continuing doing what we have been doing in recent work to design and implement solutions like Debsources, UDD, and Software Heritage. Other challenges will require going back to research work we were doing a long while ago (not discussed in this manuscript) touching the Semantic Web, ontologies, and related technologies. Others challenges yet will need leaving our comfort zone and reach out to other research communities to work together on topics that have evident synergies with the mission of Software Heritage, but require different skill sets.

It will be a novel research journey.

All things considered, it sounds like a lot of fun.

References

[1] Pietro Abate, Jaap Boender, Roberto Di Cosmo, and Stefano Zacchiroli. Strong dependencies between software components. In *ESEM 2009: 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 89–99, 2009.

[2] Pietro Abate and Roberto Di Cosmo. Predicting upgrade failures using dependency analysis. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *Workshops Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 145–150. IEEE, 2011.

[3] Pietro Abate, Roberto Di Cosmo, Louis Gesbert, Fabrice Le Fessant, Ralf Treinen, and Stefano Zacchiroli. Mining component repositories for installability issues. In *MSR 2015: The 12th Working Conference on Mining Software Repositories*, pages 24–33. IEEE, 2015.

[4] Pietro Abate, Roberto Di Cosmo, Louis Gesbert, Fabrice Le Fessant, and Stefano Zacchiroli. Using preferences to tame your package manager. In *OCaml 2014: The OCaml Users and Developers Workshop*, 2014.

[5] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Mpm: a modular package manager. In *CBSE 2011: 14th International ACM SIGSOFT Symposium on Component Based Software Engineering*, pages 179–188. ACM, 2011.

[6] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Dependency solving: a separate concern in component evolution management. *Journal of Systems and Software*, 85(10):2228–2240, October 2012.

[7] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Learning from the future of component repositories. In *CBSE 2012: 15th International ACM SIGSOFT Symposium on Component Based Software Engineering*, pages 51–60. ACM, 2012.

[8] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. A modular package manager architecture. *Information and Software Technology*, 55(2):459–474, February 2013.

[9] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Learning from the future of component repositories. *Science of Computer Programming*, 90(B):93–115, 2014.

[10] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *LICS*, pages 313–321. IEEE, 1996.

[11] Bram Adams, Christian Bird, Foutse Khomh, and Kim Moir. 1st international workshop on release engineering (RELENG 2013). In *ICSE'13*, pages 1545–1546, 2013.

[12] Aeolus Team. Aeolus Tools. https://github.com/aeolus-project/.

[13] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Error and attack tolerance of complex networks. *Nature*, 406(6794):378–382, 2000.

[14] Roberto Amadini. Evaluation and application of portfolio approaches in constraint programming. *Theory and Practice of Logic Programming (TPLP)*, 13(4-5-Online-Supplement), 2013.

[15] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An empirical evaluation of portfolios approaches for solving CSPs. In *CPAIOR 2012: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference*, volume 7874 of *Lecture Notes in Computer Science*, pages 316–324, 2013.

[16] David Anderson. The digital dark age. *Communications of the ACM*, 58(12):20–23, 2015.

[17] Josep Argelich, Daniel Le Berre, Inês Lynce, João P. Marques Silva, and Pascal Rapicault. Solving linux upgradeability problems using boolean optimization. In *LoCoCo 2010: Proceedings of the 1st International Workshop on Logics for Component Configuration*, pages 11–22, 2010.

[18] William Y. Arms. Uniform resource names: handles, purls, and digital object identifiers. *Commun. ACM*, 44(5):68, 2001.

[19] Cyrille Valentin Artho, Roberto Di Cosmo, Kuniyasu Suzaki, and Stefano Zacchiroli. Sources of inter-package conflicts in Debian. In *LoCoCo 2011 International Workshop on Logics for Component Configuration*, 2011.

[20] Cyrille Valentin Artho, Kuniyasu Suzaki, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Why do software packages conflict? In *MSR 2012: 9th IEEE Working Conference on Mining Software Repositories*, pages 141–150. IEEE, 2012.

[21] Donald Beagle. Conceptualizing an information commons. *The Journal of Academic Librarianship*, 25(2):82–89, 1999.

[22] Daniel Le Berre and Anne Parrain. On SAT technologies for dependency management and beyond. In *Software Product Lines, 12th International Conference, SPLC 2008*, pages 197–200. Lero Int. Science Centre, University of Limerick, Ireland, 2008.

[23] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[24] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data-the story so far. *Semantic services, interoperability and web applications: emerging concepts*, pages 205–227, 2009.

[25] Jaap Boender. Efficient computation of dominance in component systems (short paper). In *SEFM: Software Engineering and Formal Methods - 9th International Conference*, pages 399–406, 2011.

[26] Hongyu Pei Breivold, Muhammad Aufeef Chauhan, and Muhammad Ali Babar. A systematic review of studies of open source software evolution. In *APSEC*, pages 356–365, 2010.

[27] Brian Harry's blog. Shutting down CodePlex. https://blogs.msdn.microsoft.com/bharry/2017/03/31/shutting-down-codeplex/, 2017.

[28] Dan Brickley and Libby Miller. Foaf vocabulary specification 0.91. Technical report, ILRT, 2007.

[29] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in Java. *Softw., Pract. Exper.*, 36(11-12):1257–1284, 2006.

[30] Matthieu Caneill, Daniel M. Germán, and Stefano Zacchiroli. The Debsources dataset: Two decades of free and open source software. *Empirical Software Engineering*, 22:1405–1437, June 2017.

[31] Matthieu Caneill and Stefano Zacchiroli. Debsources: Live and historical views on macro-level software evolution. In *ESEM 2014: 8th International Symposium on Empirical Software Engineering and Measurement*. ACM, 2014.

[32] Michel Catan, Roberto Di Cosmo, Antoine Eiche, Tudor A. Lascu, Michael Lienhardt, Jacopo Mauro, Ralf Treinen, Stefano Zacchiroli, Gianluigi Zavattaro, and Jakub Zwolakowski. Aeolus: Mastering the complexity of cloud application deployment. In *ESOCC 2013: Service-Oriented and Cloud Computing*, volume 8135 of *LNCS*, pages 1–3. Springer-Verlag, 2013.

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

[34] Scott Chacon and Ben Straub. *Pro Git*. Apress, Berkely, CA, USA, 2nd edition, 2014.

[35] Antonio Cicchetti, Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio, and Stefano Zacchiroli. A model driven approach to upgrade package-based software systems. In *ENASE 2009: 4th international conference on Evaluation of Novel Aspects to Software Engineering*, pages 262–276. Springer-Verlag, 2010.

[36] Eric Clayberg and Dan Rubel. *Eclipse Plug-ins (3rd Edition)*. Addison-Wesley Professional, 3 edition, December 2008.

[37] Cloud Foundry. http://cloudfoundry.org/.

[38] Christian Collberg, Todd Proebsting, Gina Moraila, Akash Shankaran, Zuoming Shi, and Alex M Warren. Measuring reproducibility in computer systems research. Technical report, Department of Computer Science, University of Arizona, Tech. Rep, 2014.

[39] Russ Cox. Regular expression matching with a trigram index or how google code search worked. https://swtch.com/~rsc/regexp/regexp4.html, 2012.

[40] IBM ILOG Cplex. User's manual. *ILOG. See ftp://ftp. software. ibm. com/software/websphere/ilog/docs/optimization/cplex/ps_usrmancplex. pdf*, 2010.

[41] Quynh Dang. Changes in federal information processing standard (fips) 180-4, secure hash standard. *Cryptologia*, 37(1):69–73, 2013.

[42] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[43] Serge Demeyer, Alessandro Murgia, Kevin Wyckmans, and Ahmed Lamkanfi. Happy birthday! a trend analysis on past msr papers. In *MSR 13: 10th Working Conference on Mining Software Repositories*, MSR'13, pages 353–362, Piscataway, NJ, USA, 2013. IEEE.

[44] J. Des Rivières and J. Wiegand. Eclipse: a platform for integrating development tools. *IBM Systems*, 43(2):371–383, 2004.

[45] Roberto Di Cosmo, Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio, and Stefano Zacchiroli. Supporting software evolution in component-based FOSS systems. *Science of Computer Programming*, 76(12):1144–1160, 2011.

[46] Roberto Di Cosmo, Antoine Eiche, Jacopo Mauro, Stefano Zacchiroli, Gianluigi Zavattaro, and Jakub Zwolakowski. Automatic deployment of services in the cloud with aeolus blender. In *ICSOC 2015: 13th International Conference on Service Oriented Computing*, pages 397–411. Springer-Verlag, 2015.

[47] Roberto Di Cosmo, Olivier Lhomme, and Claude Michel. Aligning component upgrades. In Conrad Drescher, Inês Lynce, and Ralf Treinen, editors, *LoCoCo 2011 International Workshop on Logics for Component Configuration*, volume 65, pages 1–11, 2011.

[48] Roberto Di Cosmo, Michael Lienhardt, Jacopo Mauro, Stefano Zacchiroli, Gianluigi Zavattaro, and Jakub Zwolakowski. Automatic application deployment in the cloud: from practice to theory and back. In *CONCUR 2015: 26th International Conference on Concurrency Theory*, volume 42 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1–16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.

[49] Roberto Di Cosmo, Michael Lienhardt, Ralf Treinen, Stefano Zacchiroli, and Jakub Zwolakowski. Optimal provisioning in the cloud. Technical report, Aeolus project, Juin 2013.

[50] Roberto Di Cosmo, Michael Lienhardt, Ralf Treinen, Stefano Zacchiroli, Jakub Zwolakowski, Antoine Eiche, and Alexis Agahi. Automated synthesis and deployment of cloud applications. In *ASE 2014: 29th IEEE/ACM International Conference on Automated Software Engineering*, pages 211–222. ACM, 2014.

[51] Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, and Gianluigi Zavattaro. Component reconfiguration in the presence of conflicts. In *ICALP 2013: 40th International Colloquium on Automata, Languages and Programming*, volume 7966 of *LNCS*, pages 187–198. Springer-Verlag, 2013.

[52] Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, and Gianluigi Zavattaro. Aeolus: a component model for the cloud. *Information and Computation*, 239:100–121, 2014.

[53] Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Formal aspects of free and open source software components. In *FMCO 2012: HATS International School on Formal Models for Components and Objects*, volume 7866 of *LNCS*, pages 216–239. Springer-Verlag, 2013.

[54] Roberto Di Cosmo, Paulo Trezentos, and Stefano Zacchiroli. Package upgrades in FOSS distributions: Details and challenges. In *HotSWUp'08: Hot Topics in Software Upgrades*. ACM, 2008.

[55] Roberto Di Cosmo and Jérôme Vouillon. On software component co-installability. In *SIGSOFT FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19)*, pages 256–266. ACM, 2011.

[56] Roberto Di Cosmo and Stefano Zacchiroli. Feature diagrams as package dependencies. In *SPLC 2010: 14th International Software Product Line Conference*, volume 6287 of *LNCS*, pages 476–480. Springer-Verlag, 2010.

[57] Roberto Di Cosmo and Stefano Zacchiroli. Software heritage: Why and how to preserve software source code. In *iPRES 2017: 14th International Conference on Digital Preservation*, 2017.

[58] Roberto Di Cosmo, Stefano Zacchiroli, and Gianluigi Zavattaro. Towards a formal component model for the cloud. In *SEFM 2012: 10th International Conference on Software Engineering and Formal Methods*, volume 7504 of *LNCS*, pages 156–171. Springer-Verlag, 2012.

[59] Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio, and Stefano Zacchiroli. Towards maintainer script modernization in FOSS distributions. In *IWOCE 2009: International Workshop on Open Component Ecosystem*, pages 11–20. ACM, 2009.

[60] Edd Dumbill. Doap: Description of a project. https://github.com/ewilderj/doap, 2010.

[61] Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488. ACM, 2014.

[62] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[63] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 422–431. IEEE Press, 2013.

[64] E-ark (european archival records and knowledge preservation) project. http://www.eark-project.com/, 2014.

[65] EDOS Project. Report on formal management of software dependencies. EDOS Project Deliverables D2.1 and D2.2, EDOS Project, March 2006.

[66] Keep: Eu cooperating. https://www.keep.eu/, 2000.

[67] Ingo Feinerer. Efficient large-scale configuration via integer linear programming. *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing (AI EDAM)*, 27(1):37–49, 2013.

[68] Ingo Feinerer and Gernot Salzer. Consistency and minimality of UML class specifications with multiplicities and uniqueness constraints. In *Theoretical Aspects of Software Engineering (TASE)*, pages 411–420, 2007.

[69] Joseph Feller, Brian Fitzgerald, et al. *Understanding open source software development*. Addison-Wesley London, 2002.

[70] A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256:63–92, 2001.

[71] Jeffrey Fischer, Rupak Majumdar, and Shahram Esmaeilsabzali. Engage: a deployment management system. In *PLDI'12: Programming Language Design and Implementation*, pages 263–274. ACM, 2012.

[72] Martin Gebser, Roland Kaminski, and Torsten Schaub. aspcud: A linux package configuration tool based on answer set programming. *arXiv preprint arXiv:1109.0113*, 2011.

[73] Daniel M. German, Yuki Manabe, and Katsuro Inoue. A sentence-matching method for automatic license identification of source code files. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE'10, pages 437–446. ACM, 2010.

[74] Carlo Ghezzi. Reflections on 40+ years of software engineering research and beyond: an insider's view. In *Keynote address in 31st International Conference on Software Engineering*, 2009.

[75] Inc. GitHub. Open source survey. http://opensourcesurvey.org/2017/, 2017.

[76] GitLab. Gitlab acquires gitorious to bolster its on premises code collaboration platform. https://about.gitlab.com/2015/03/03/gitlab-acquires-gitorious/, 2015.

[77] Gmane. http://gmane.org, 2017.

[78] GNU. GNU General Public License, version 2, 1991. retrieved September 2015.

[79] Robert Gobeille. The fossology project. In *MSR 2008: The 5th Working Conference on Mining Software Repositories*, pages 47–50. ACM, 2008.

[80] Jesús M. González-Barahona, Gregorio Robles, Martin Michlmayr, Juan José Amor, and Daniel M. Germán. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285, 2009.

[81] Ramanathan V Guha, Dan Brickley, and Steve Macbeth. Schema.org: Evolution of structured data on the web. *Communications of the ACM*, 59(2):44–51, 2016.

[82] Ahmed E Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 48–57. IEEE, 2008.

[83] Brian Hayes. Cloud computing. *Communications of the ACM*, 51:9–11, 2008.

[84] John A. Hewson, Paul Anderson, and Andrew D. Gordon. A Declarative Approach to Automated Configuration. In *LISA '12: Large Installation System Administration Conference*, pages 51–66, 2012.

[85] Roger Highfield. Large hadron collider: Thirteen ways to change the world. *The Daily Telegraph*, October 2008.

[86] Abram Hindle, Israel Herraiz, Emad Shihab, and Zhen Ming Jiang. Mining challenge 2010: Freebsd, gnome desktop and debian/ubuntu. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 82–85. IEEE, 2010.

[87] Google Project Hosting. Bidding farewell to google code. https://opensource.googleblog.com/2015/03/farewell-to-google-code.html, 2015.

[88] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[89] Internet archive: Digital library of free books, movies, music & wayback machine. https://archive.org, 2017.

[90] Mikolás Janota. Do SAT solvers make good configurators? In *SPLC: Software Product Lines Conference, 2nd Volume*, pages 191–195, 2008.

[91] Nicolas Jeannerod, Claude Marché, and Ralf Treinen. A Formally Verified Interpreter for a Shell-like Programming Language. In *VSTTE 2017 - 9th Working Conference on Verified Software: Theories, Tools, and Experiments*, Heidelberg, Germany, July 2017.

[92] Nicolas Jeannerod, Yann Régis-Gianas, and Ralf Treinen. Having Fun With 31.521 Shell Scripts. working paper or preprint, April 2017.

[93] Graham Jenson, Jens Dietrich, and Hans W. Guesgen. An empirical study of the component dependency resolution search space. In *CBSE 2011: International ACM Sigsoft Symposium on Component Based Software Engineering*, volume 6092 of *LNCS*, pages 182–199. Springer, 2010.

[94] Jones, Matthew B., Carl Boettiger, Abby Cabunoc Mayes, Arfon Smith, Peter Slaughter, Kyle Niemeyer, Yolanda Gil, Martin Fenner, Krzysztof Nowak, Mark Hahnel, Luke Coy, Alice Allen, Mercè Crosas, Ashley Sands, Neil Chue Hong, Patricia Cruse, Dan Katz, and Carole Goble. CodeMeta: an exchange schema for software metadata. version 2.0. https://codemeta.github.io/, 2017.

[95] Juju, devops distilled. https://juju.ubuntu.com/.

[96] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of software maintenance and evolution: Research and practice*, 19(2):77–131, 2007.

[97] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[98] Luke Kanies. Puppet: Next-generation configuration management. *;login: the USENIX magazine*, 31(1):19–25, 2006.

[99] Michael Kerrisk. Surveying open source licenses. Available at https://lwn.net/Articles/547400/, 2013.

[100] Nancy Kranich and Jorge Reina Schement. Information commons. *Annual Review of Information Science and Technology*, 42(1):546–591, 2008.

[101] Alyson La. Language trends on GitHub. Available at https://github.com/blog/2047-language-trends-on-github, 2015.

[102] Nathan LaBelle and Eugene Wallingford. Inter-package dependency networks in open-source software. *CoRR*, cs.SE/0411096, 2004.

[103] Ralf Lämmel, Rocco Oliveto, and Romain Robbes, editors. *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*. IEEE Computer Society, 2013.

[104] Tudor A. Lascu, Jacopo Mauro, and Gianluigi Zavattaro. Automatic component deployment in the presence of circular dependencies. In *Formal Aspects of Component Software - 10th International Symposium, FACS 2013*, volume 8348 of *Lecture Notes in Computer Science*, pages 254–272. Springer, 2013.

[105] Tudor A. Lascu, Jacopo Mauro, and Gianluigi Zavattaro. A planning tool supporting the deployment of cloud applications. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, pages 213–220, 2013.

[106] Daniel Le Berre. *Sat4j: un moteur libre de raisonnement en logique propositionnelle*. HDR (habilitation à diriger des recherches) thesis, Université d'Artois, 2010.

[107] Daniel Le Berre and Pascal Rapicault. Dependency management for the Eclipse ecosystem: Eclipse P2, metadata and resolution. In *Proceedings of the 1st international workshop on Open component ecosystems*, IWOCE '09, pages 21–30, New York, 2009. ACM.

[108] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[109] Josh Lerner and Jean Tirole. Some simple economics of open source. *The journal of industrial economics*, 50(2):197–234, 2002.

[110] Xavier Leroy, Damien Doligez, Jacques Garrigue, and Didier Rémy. *The Objective Caml system release 4.01; Documentation and user's manual*. INRIA, Rocquencourt, Paris, 2013.

[111] Gloriana St Clair Mahadev Satyanarayanan, Benjamin Gilbert, Yoshihisa Abe, Jan Harkes, Dan Ryan, Erika Linke, and Keith Webster. One-click time travel. Technical report, Technical report, Computer Science, Carnegie Mellon University, 2015.

[112] T. Maillart, D. Sornette, S. Spaeth, and G. von Krogh. Empirical tests of zipf's law mechanism in open source linux distribution. *Phys. Rev. Lett.*, 101:218701, Nov 2008.

[113] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[114] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jérôme Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the complexity of large free and open source package-based software distributions. In *ASE 2006*, pages 199–208. IEEE, 2006.

[115] Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the zinc modelling language. *Constraints*, 13(3):229–267, 2008.

[116] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 369–378. Springer, 1987.

[117] Claude Michel and Michel Rueher. Handling software upgradeability problems with MILP solvers. In *LoCoCo 2010: Proceedings of the 1st International Workshop on Logics for Component Configuration*, pages 1–10, 2010.

[118] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming (CP)*, pages 529–543, 2007.

[119] Lucas Nussbaum and Stefano Zacchiroli. The ultimate Debian database: Consolidating bazaar metadata for quality assurance and data mining. In *MSR 2010: 7th IEEE Working Conference on Mining Software Repositories*, pages 52–61. IEEE, 2010.

[120] Jonathan Oliver, Chun Cheng, and Yanggui Chen. Tlsh - a locality sensitive hash. In *CTC, 4th Cybercrime and Trustworthy Computing Workshop*, pages 7–13. IEEE, 2013.

[121] Openaire. https://www.openaire.eu/, 2014.

[122] Opscode. Chef. http://www.opscode.com/chef/.

[123] OSGi Alliance. *OSGi Service Platform, Release 3.* IOS Press, Inc., 2003.

[124] Bryan O'Sullivan. Making sense of revision-control systems. *Communications of the ACM*, 52(9):56–62, 2009.

[125] Yoann Padioleau, Julia L Lawall, and Gilles Muller. Understanding collateral evolution in linux device drivers. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 59–71. ACM, 2006.

[126] Norman Paskin. Digital object identifier (doi) system. *Encyclopedia of library and information sciences*, 3:1586–1592, 2010.

[127] Ken Pepple. *Deploying openstack.* O'Reilly Media, Inc., 2011.

[128] Martin Pinzger, Gabriele Bavota, and Andrian Marcus, editors. *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*. IEEE Computer Society, 2017.

[129] Clément Quinton, Romain Rouvoy, and Laurence Duchien. Leveraging feature models to configure virtual appliances. In *Proceedings of the 2nd International Workshop on Cloud Computing Platforms*, CloudCP '12, pages 2:1–2:6, New York, NY, USA, 2012. ACM.

[130] Erhard Rahm and Philip A Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal—The International Journal on Very Large Data Bases*, 10(4):334–350, 2001.

[131] Eric S. Raymond. *The cathedral and the bazaar*. O'Reilly, 2001.

[132] Eric S Raymond. *The art of Unix programming*. Addison-Wesley Professional, 2003.

[133] Gregorio Robles, Jesus M Gonzalez-Barahona, and Martin Michlmayr. Evolution of volunteer participation in libre software projects: evidence from debian. In *Proceedings of the 1st International Conference on Open Source Systems*, pages 100–107, 2005.

[134] David Rosenthal, Rob Baxter, and Laurence Field. Towards a shared vision of sustainability for research and e-infrastructures. https://www.eudat.eu/news/towards-shared-vision-sustainability-research-and-e-infrastructures, 24-25 September 2014. EUDAT conference.

[135] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 410–424. ACM, 2015.

[136] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[137] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.

[138] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 1157–1168. IEEE, 2016.

[139] Peter H Salus. *A quarter century of UNIX*. Addison-Wesley Reading, 1994.

[140] Philippe Schnoebelen. Revisiting ackermann-hardness for lossy counter machines and reset petri nets. In *MFCS*, volume 6281 of *Lecture Notes in Computer Science*, pages 616–628. Springer, 2010.

[141] Christian Schulte, Mikael Lagerkvist, and Guido Tack. Gecode. http://www.gecode.org/.

[142] Charles M Schweik and Robert C English. *Internet success: a study of open-source software commons*. MIT Press, 2012.

[143] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable SCA applications with the FraSCAti platform. In *IEEE SCC: International Conference on Services Computing*, pages 268–275, 2009.

[144] Megan Squire and David Williams. Describing the software forge ecosystem. In *System Science (HICSS), 2012 45th Hawaii International Conference on*, pages 3416–3425. IEEE, 2012.

[145] Richard Stallman. *Free software, free society: Selected essays of Richard M. Stallman*. GNU Press, 2002.

[146] Michael Stapelberg. Debian Code Search. B.S. thesis, Hochschule Mannheim, 2012.

[147] Ralph E. Steuer. *Multiple Criteria Optimization: Theory, Computation and Application*. Wiley, 1986.

[148] Kate Stewart, Phil Odence, and Esteban Rockett. Software package data exchange (SPDX™) specification. *International Free and Open Source Software Law Review*, 2(2):191–196, 2011.

[149] Michael Stonebraker and Lawrence A. Rowe. The design of postgres. *SIGMOD Rec.*, 15(2):340–355, June 1986.

[150] Peter J. Stuckey, Maria Garcia de la Banda, Michael Maher, John Slaney, Zoltan Somogyi, Mark Wallace, and Toby Walsh. The G12 project: Mapping solver independent models to efficient solutions. In *ICLP 2005: Logic Programming, 21st International Conference*, volume 3668 of *Lecture Notes in Computer Science*, pages 9–13, 2005.

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

[152] Clemens Szyperski. *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[153] Ralf Treinen and Stefano Zacchiroli. Solving package dependencies: from EDOS to Mancoosi, 2008.

[154] Ralf Treinen and Stefano Zacchiroli. Common upgradeability description format (cudf) 2.0. Technical report, Mancoosi project, November 2009.

[155] Ralf Treinen and Stefano Zacchiroli. Expressing advanced user preferences in component installation. In *IWOCE 2009: International Workshop on Open Component Ecosystems*, pages 31–40. ACM, 2009.

[156] Ralf Treinen and Stefano Zacchiroli. Expressing advanced user preferences in component installation. In *IWOCE 2009: International Workshop on Open Component Ecosystem*, pages 31–40. ACM, 2009.

[157] Paulo Trezentos, Inês Lynce, and Arlindo Oliveira. Apt-pbo: Solving the software dependency problem using pseudo-boolean optimization. In *ASE'10: Automated Software Engineering*, pages 427–436. ACM, 2010.

[158] V Trimble and JA Ceja. Productivity and impact of astronomical facilities: A recent sample. *Astronomische Nachrichten*, 331(3):338–345, 2010.

[159] Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. OPIUM: Optimal package install/uninstall manager. In *ICSE'07: International Conference on Software Engineering*, pages 178–188. IEEE, 2007.

[160] Frederic Tuong, Fabrice Le Fessant, and Thomas Gazagnaire. OPAM: an OCaml package manager. In *SIGPLAN OCaml Users and Developers Workshop*, 2012.

[161] Unesco persist programme. https://unescopersist.org/, 2015.

[162] Herbert Van de Sompel and Andrew Treolar. A perspective on archiving the scholarly web. *Proceedings of the International Conference on Digital Preservation (iPRES)*, pages 194–198, 2014.

[163] Richard Van Noorden, Brendan Maher, and Regina Nuzzo. The top 100 papers. *Nature*, pages 550–553, October4 2014.

[164] Jérôme Vouillon, Mehdi Dogguy, and Roberto Di Cosmo. Easing software component repository evolution. In *Proceedings of the 36th International Conference on Software Engineering*, pages 756–766. ACM, 2014.

[165] Steve Weber. *The success of open source*. Harvard University Press, 2004.

[166] Joel West and Scott Gallagher. Challenges of open innovation: the paradox of firm investment in open-source software. *R&d Management*, 36(3):319–331, 2006.

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

[168] Stefano Zacchiroli. The debsources dataset: Two decades of Debian source code metadata. In *MSR 2015: The 12th Working Conference on Mining Software Repositories*, pages 466–469. IEEE, 2015.

[169] Zenodo. https://zenodo.org/, 2013.

[170] Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. Searching large lexicons for partially specified terms using compressed inverted files. In *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 290–301. Morgan Kaufmann Publishers Inc., 1993.

[171] Jakub Zwolakowski. *A formal approach to distributed application synthesis and deployment automation*. PhD thesis, University Paris Diderot, France, 2015.

Part II

Curriculum Vitae

Chapter 7

Detailed curriculum vitae and list of publications

Due to local requirements this chapter is in French. For the most part it covers facts about scientific artifacts that are reported using their original (English) names.

Données personnelles

Stefano Zacchiroli
né le 16 mars 1979 à Bologne, Italie
marié, un enfant

Adresse professionnelle:

Laboratoire IRIF
bureau 3019
Bâtiment Sophie Germain
8 place Aurélie Nemours
75013 Paris

Email: zack@upsilon.cc
Page web: http://upsilon.cc/zack
Fingerprint GPG: 4900 707D DC5C 07F2 DECB 0283 9C31 503C 6D86 6396

Langues

Italien langue maternelle (ILR 5—*native or bilingual proficiency*)

Français lu, écrit et parlé couramment (ILR 4—*full professional proficiency*)

Anglais lu, écrit et parlé couramment (ILR 4—*full professional proficiency*)

Espagnol compréhension de base (ILR 1—*elementary proficiency*)

Emplois dans l'université et la recherche

2011–présent maître de conférences en informatique, Laboratoire IRIF, Université Paris Diderot

2011 post-doc en informatique, Laboratoire PPS, Université Paris Diderot, projet Aeolus; directeur: Prof. Roberto Di Cosmo

2008–2011 post-doc en informatique, Laboratoire PPS, Université Paris Diderot, projet Mancoosi; directeur: Prof. Roberto Di Cosmo

2007 post-doc en informatique, Département d'Informatique, Université de Bologne, Italie; directeur: Prof. Fabio Vitali

2006 chercheur invité, Département d'Informatique, Université Brown, Providence, RI, États-Unis; directeur: Prof. David G. Durand

2004–2007 titulaire de contrat de recherche (*assegno di ricerca*), Département d'Informatique, Université de Bologne, Italie

Formation

2007 **Doctorat en Informatique**, Université de Bologne (Italie). Thèse effectuée au Département d'Informatique. Titre: *User Interaction Widgets for Interactive Theorem Proving*. Directeur: Andrea Asperti. Jury: Christoph Benzmueller, Marino Miculan, Roberto Giacobazzi, Simonetta Balsamo, Gianluigi Ferrari, Giovanni Pau.

2003 ***Laurea*** (équivalent Master 2), Université de Bologne (Italie). Dissertation effectuée au Département d'Informatique. Titre: *Web services per il supporto alla dimostrazione interattiva*. Directeur: Andrea Asperti. Mention: 110/110, *cum laude*.

Recherche

Comités des conférences, workshops, revues

Revues internationales

2015-présent membre du comité de lecture du blog de la revue *IEEE Software* (http://blog.ieeesoftware.org/)

2013-présent membre du comité de lecture de la revue *Journal of Peer Production* (http://peerproduction.net/)

2013 membre du comité de lecture de la revue *Journal of Web Engineering*, Rinton Press, *special issue* au sujet des technologies web (http://www.rintonpress.com/journals/jwe/

2012 membre du comité de lecture de la revue *Science of Computer Programming*, Elsevier, *special issue* au sujet des technologies web (http://www.journals.elsevier.com/science-of-computer-programming)

2011 membre du comité de lecture de la revue *Software: Practice and Experience*, Wiley, *special issue* au sujet des technologies web (http://onlinelibrary.wiley.com/journal/10.1002/(ISSN)1097-024X)

Conférences et workshops internationaux

2017 *free/open source software chair* de MSR 2017 (International Conference on Mining Software Repositories)

2017 *proceedings chair* de OSS 2017 (International Conference on Open Source Systems)

2013,2015–présent membre du comité de programme de OpenSym (International Symposium on Open Collaboration) http://www.opensym.org/

2014–présent membre du comité de programme de OSS (International Conference on Open Source Systems)

2013–présent membre du comité de programme de RelEng (International Workshop on Release Engineering) http://releng.polymtl.ca/

2011–2015 co-président du comité de programme de SAC WT (ACM Symposium on Applied Computing, track Web Technologies) http://www.acm.org/conferences/sac/

2012–2013 membre du comité de programme de OSDOC (Workshop Open Source and Design of Communication)

2010 membre du comité de programme de SAC WT (ACM Symposium on Applied Computing, track Web Technologies) http://www.acm.org/conferences/sac/

2010 membre du comité d'organisation de la compétition MISC 2010: *Mancoosi International Solver Competition*

2009 *publicity chair* de IWOCE 2009 (International Workshop on Open Component Ecosystems)

Conférences et workshops nationaux

2009–2010,2012,2015–présent membre du comité de programme de CONFSL (conférence italienne sur le logiciel libre) http://www.confsl.it/

2015 membre du comité de programme de SCORE-it 2015 (Italian Student Contenst in Software Engineering)

Rapporteur

En dehors de ma participation dans des comités de programme, pendant les 5 derniers années j'ai aussi été rapporteur pour les revues internationales suivantes:[1]

- IEEE Software
- Information Technology (De Gruyter Open)
- Journal of Systems and Software (Elsevier)
- PeerJ
- Science of Computer Programming (Elsevier)
- Software: Practice and Experience (Wiley)

Thèses J'ai été rapporteur pour les thèses de doctorat suivantes:

2016 Jacopo Soldani, *Modelling, analysing, and reusing composite cloud applications*, Université de Pise (Italie), dirigé par Antonio Brogi

[1] https://publons.com/author/705657/stefano-zacchiroli#stats

Conseils scientifiques et responsabilités collectives

2017–présent membre du conseil scientifique, projet européen Horizon 2020 CROSS-MINER (https://www.crossminer.org/), grant No. 732223

2016–présent membre du conseil scientifique, UFR Informatique, Université Paris Diderot

2016–présent membre du conseil consultatif, association *Center for the Cultivation of Technology*, Germany (https://techcultivation.org)

2016–présent membre du conseil consultatif, entreprise Purism, USA (https://puri.sm)

2013–présent membre du conseil consultatif, entreprise Bitergia, Spain (https://bitergia.com)

2013–présent membre du comité de déontologie, association Nos oignons, France (https://nos-oignons.net)

2014–2017 membre du conseil d'administration, association Open Source Initiative, USA (http://www.opensource.org)

2010–2013 *Debian Project Leader*, projet Debian (http://www.debian.org)

2013 membre du groupe de travail de l'agence publique pour le numérique du gouvernement italien (*Agenzia per l'Italia Digitale*) sur les critères d'acquisition des logiciels dans l'administration publique

2012 membre du comité scientifique pour l'agenda numérique de la ville de Bologne (Italie)

2011 membre du group de travail W3C *Social Web Incubator* (http://www.w3.org/2005/Incubator/socialweb)

2007–2011 membre du group de travail W3C XML Schema (http://www.w3.org/XML/Schema)

2007–2011 membre du conseil scientifique du Master en Technologies du Logiciel Libre, Université de Bologne (Italie)

Distinctions

2014–2018 titulaire de la PEDR (Prime d'Encadrement Doctoral et de Recherche) 2014–2018

2015 récompensé par le prix *O'Reilly Open Source Award*[2]

2014 récompensé avec un *Flash Grant* de la fondation Shuttleworth[3]

2012 *Best Paper Award* pour l'article « Learning from the Future of Component Repositories », CBSE 2012: 15th International ACM SIGSOFT Symposium on Component Based Software Engineering

2011 *Distinguished Paper Award* pour l'article « MPM: a modular package manager », CBSE 2011: 14th International ACM SIGSOFT Symposium on Component Based Software Engineering

[2] https://en.wikipedia.org/wiki/O%27Reilly_Open_Source_Award
[3] https://www.shuttleworthfoundation.org/fellows/flash-grants/

Participation à des projets de recherche

2016–présent co-fondateur et CTO (*Chief Technology Officier*) du projet Inria Software Heritage (https://www.softwareheritage.org)

2016–présent participant au projet ANR Colis: « Correctness of Linux Scripts »

2010–2014 responsable du site Université Paris Diderot, projet ANR Aeolus: « Maîtriser la complexité du Cloud Computing »

2010–2013 participant au projet de recherche FEDER DORM: « Derived Objects Repository Manager », projet financé par le pôle de compétitivité Île de France, Groupe Thématique Logiciel Libre

2008–2011 participant au projet de recherche européen FP7 Mancoosi: « MANaging the COmplexity of the Open Source Infrastructure »

2005–2007 participant au projet de recherche européen (IST *working group*) TYPES (http://www.cse.chalmers.se/research/group/logic/Types)

2005–2006 participant au projet de recherche PRIN (programme de recherche national italien, eq. ANR) McTAFI: « Méthodes Constructives pour la Topologie Algébrique »

2002–2006 participant au projet de recherche européen FP6 MoWGLI: « Mathematics on the Web: Get It by Logics and Interfaces »

2003 participant au projet de recherche européen FP5 MKM-NET: « Mathematical Knowledge Management Network »

Enseignement

2015–présent cours, Conduite de projet, Licence 3 en informatique, Université Paris Diderot

2014–présent cours et TD/TP, Logiciels libres, Master 1 en informatique, Université Paris Diderot

2011–2015 cours, Programmation système, Master 1 en informatique, Université Paris Diderot

2011–2015 cours et TD, Génie Logiciel Avancé, Master 1 en Informatique, Université Paris Diderot

2013–2014 cours, Méthodes de test, Master 2 en informatique, Université Paris Diderot

2013–2014 cours, Logiciels libres, Licence 2 (tous parcours scientifiques), Université Paris Diderot

2012–2014 cours, Programmation fonctionnelle, Licence 3 en informatique, Université Paris Diderot

2012–2014 cours, Projet long, Master 1 en informatique, Université Paris Diderot

2011-2014 cours, Environnements et outils de développement, Licence 3 en informatique, Université Paris Diderot

2011–2014 TP, Programmation objet: concepts avancés, Master 2 en informatique, Université Paris Diderot

2008–2010 cours, *Basi di Dati e Programmazione Web* (bases de données et programmation web), Master en Technologies du Logiciel Libre, Université de Bologne, Italie

2007–2010 cours, *Logica Matematica* (logique mathématique), apprentissage en ligne, Licence 1 en informatique, Université de Urbino, Italie

2004–2008 TP, *Programmazione* (Programmation), Licence 1 en informatique, Université de Bologne, Italie

2006–2007 cours, *Laboratorio di Sistemi Informativi* (laboratoire des bases de données), Master en Technologies du Logiciel Libre, Université de Bologne, Italie

2004–2007 cours, *Laboratorio di Sistemi Operativi* (laboratoire des systèmes d'exploitation), Master en Technologies du Logiciel Libre, Université de Bologne, Italie

Encadrement

Thèses de doctorat

2017–présent co-direction de thèse, Antoine Pietri, Inria

2011–2014 co-direction de thèse, Jakub Zwolakowski, Université Paris Diderot

Stages

2017 direction de stage Master 2, Morane Gruenpeter, Inria

2017 direction de stage Master 2, Sushant Mukhija, Inria

2016 direction de stage Master 2, Quentin Campos, Inria

2016 direction de stage Master 1, Jordi Bertran De Balanda, Inria

2015 direction de stage Licence 3, Clément Schreiner, Inria

2014 direction de stage Master 2 (*tesi di laurea*), Giacomo Mantani, Université de Bologne (Italie)

2013 direction de stage Master 1, Matthieu Caneill, Inria

2008 direction de stage Master 2 (*tesi di laurea*), Edoardo Gargano, Université de Bologne (Italie)

2006 direction de stage Master 2 (*tesi di laurea*), Paolo Marinelli, Université de Bologne (Italie)

Autres

2012–2013 co-encadrement post-doc, Michael Lienhardt, projet ANR Aeolus, Université Paris Diderot

2015 encadrement Google Summer of Code, Orestis Ioannou, projet Debian

2014 encadrement FOSS Outreach Program, Jingjie Jiang, projet Debian

2014 encadrement Google Summer of Code, Joseph Bisch, projet Debian

2013 encadrement Google Summer of Code, Boris Bobrov, projet Debian

2008 encadrement Google Summer of Code, Christian von Essen, projet Debian

2007 encadrement Google Summer of Code, Margerita Manterola, projet Debian

Publications

Revues internationales, avec comité de lecture

1. *The Debsources Dataset: Two Decades of Free and Open Source Software* with Matthieu Caneill, Daniel M. Germán. In Empirical Software Engineering, Volume 22, pp. 1405-1437, June, 2017. ISSN 1382-3256, Springer.
DOI 10.1007/s10664-016-9461-5.
2. *Aeolus: a Component Model for the Cloud* with Roberto Di Cosmo, Jacopo Mauro, Gianluigi Zavattaro. In Information and Computation, Volume 239, pp. 100-121. 2014. ISSN 0890-5401, Elsevier.
DOI 10.1016/j.ic.2014.11.002.
3. *Learning from the Future of Component Repositories* with Pietro Abate, Roberto Di Cosmo, Ralf Treinen. In Science of Computer Programming, Volume 90, Part B, pp. 93-115. ISSN 0167-6423, Elsevier, 2014.
DOI 10.1016/j.scico.2013.06.007.
4. *A Modular Package Manager Architecture* with Pietro Abate, Roberto Di Cosmo, Ralf Treinen. In Information and Software Technology, Volume 55, Issue 2, pp. 459-474. ISSN 0950-5849, Elsevier, February 2013.
DOI 10.1016/j.infsof.2012.09.002.
5. *Dependency Solving: a Separate Concern in Component Evolution Management* with Pietro Abate, Roberto Di Cosmo, Ralf Treinen. In Journal of Systems and Software, Volume 85, Issue 10, pp. 2228-2240. ISSN 0164-1212, Elsevier, October 2012.
DOI 10.1016/j.jss.2012.02.018.
6. *Constrained Wiki: The WikiWay to Validating Content* with Angelo Di Iorio, Francesco Draicchio, Fabio Vitali. In Advances in Human-Computer Interaction, Volume 2012, Article ID 893575, pp. 1-19. Hindawi, 2012
DOI 10.1155/2012/893575.
7. *Supporting Software Evolution in Component-Based FOSS Systems* with Roberto Di Cosmo, Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio. In Science of Computer Programming, Volume 76, Issue 12, pp. 1144-1160. ISSN 0167-6423, Elsevier, 2011.
DOI 10.1016/j.scico.2010.11.001.
8. *Towards the unification of formats for overlapping markup* with Paolo Marinelli, Fabio Vitali. In New Review of Hypermedia and Multimedia, Volume 14, Issue 1, January 2008, pp. 57-94. Taylor and Francis, ISSN 1361-4568.
DOI 10.1080/13614560802316145.
9. *Spurious Disambiguation Errors and How to Get Rid of Them* with Claudio Sacerdoti Coen. In Mathematics in Computer Science, Volume 2, Number 2, pp. 355-378, December 2008. Springer Birkhäuser, ISSN 1661-8270.
DOI 10.1007/s11786-008-0058-2.
10. *User Interaction with the Matita Proof Assistant* with Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi. In Journal of Automated Reasoning, Volume 39, Number 2. Springer Netherlands, ISSN 0168-7433, pp. 109-139, 2007.
DOI 10.1007/s10817-007-9070-5.

Éditorials

1. *Editorial* with Angelo Di Iorio, Davide Rossi. In Journal of Web Engineering, Volume 14, Number 1-2, pp. 1-2. ISSN 1540-9589, Rinton Press, 2014.
2. *Web Technologies: Selected and extended papers from WT ACM SAC 2012* with Angelo Di Iorio, Davide Rossi. In Science of Computer Programming, Volume 94, Part 1, pp. 1-2. ISSN 0167-6423, Elsevier, 2014.
   DOI `10.1016/j.scico.2014.03.001`.
3. *Editorial* with Angelo Di Iorio, Davide Rossi. In Software: Practice and Experience, Volume 43, Issue 12, pp. 1393-1394. ISSN 1097-024X, Wiley, 2013.
   DOI `10.1002/spe.2169`.

Chapitres de livre

1. *Web Semantics via Wiki Templating* with Angelo Di Iorio, Fabio Vitali. Chapter 34 of Handbook of research on Web 2.0, 3.0 and x.0: technologies, business and social applications. San Murugesan Ed., Information Science Reference, November 2009, ISBN 978-1605663845.

Conférences internationales, avec comité de lecture

1. *Software Heritage: Why and How to Preserve Software Source Code* with Roberto Di Cosmo. To appear in Proceedings of iPRES 2017: 14th International Conference on Digital Preservation, Kyoto, Japan, September 2017, 10 pages.
2. *Automatic Deployment of Services in the Cloud with Aeolus Blender* with Roberto Di Cosmo, Antoine Eiche, Jacopo Mauro, Gianluigi Zavattaro, Jakub Zwolakowski. In proceedings of ICSOC 2015: 13th International Conference on Service Oriented Computing, November 16-19, 2015, Goa, India. ISBN 978-3-662-48615-3, pp. 397-411, Springer-Verlag 2015.
   DOI `10.1007/978-3-662-48616-0_28`.
3. *Automatic Application Deployment in the Cloud: from Practice to Theory and Back* with Roberto Di Cosmo, Michael Lienhardt, Jacopo Mauro, Gianluigi Zavattaro, Jakub Zwolakowski. In proceedings of CONCUR 2015: 26th International Conference on Concurrency Theory, September 1-4, 2015, Madrid, Spain. Leibniz International Proceedings in Informatics (LIPIcs) 42, pp. 1-16, ISBN 978-3-939897-91-0, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik 2015.
   DOI `10.4230/LIPIcs.CONCUR.2015.1`.
4. *The Debsources Dataset: Two Decades of Debian Source Code Metadata* with . In proceedings of MSR 2015: The 12th Working Conference on Mining Software Repositories, May 16-17, 2015, Florence, Italy. Co-located with ICSE 2015. ISBN ISBN 978-0-7695-5594-2, pp. 466-469, IEEE 2015.
   DOI `10.1109/MSR.2015.65`.
5. *Mining Component Repositories for Installability Issues* with Pietro Abate, Roberto Di Cosmo, Louis Gesbert, Fabrice Le Fessant, Ralf Treinen. In proceedings of MSR

2015: The 12th Working Conference on Mining Software Repositories, May 16-17, 2015, Florence, Italy. Co-located with ICSE 2015. ISBN ISBN 978-0-7695-5594-2, pp. 24-33, IEEE 2015.
DOI 10.1109/MSR.2015.10.

6. *Automated Synthesis and Deployment of Cloud Applications* with Roberto Di Cosmo, Michael Lienhardt, Ralf Treinen, Jakub Zwolakowski, Antoine Eiche, Alexis Agahi. In proceedings of ASE 2014: 29th IEEE/ACM International Conference on Automated Software Engineering, September 15-19, 2014, Vasteras, Sweden. ISBN 978-1-4503-3013-8, pp. 211-222, ACM 2014.
DOI 10.1145/2642937.2642980.

7. *Debsources: Live and Historical Views on Macro-Level Software Evolution* with Matthieu Caneill. In proceedings of ESEM 2014: 8th International Symposium on Empirical Software Engineering and Measurement, September 18-19, 2014, Torino, Italy. ISBN 978-1-4503-2774-9, ACM 2014.
DOI 10.1145/2652524.2652528.

8. *Aeolus: Mastering the Complexity of Cloud Application Deployment* with Michel Catan, Roberto Di Cosmo, Antoine Eiche, Tudor A. Lascu, Michael Lienhardt, Jacopo Mauro, Ralf Treinen, Gianluigi Zavattaro, Jakub Zwolakowski. In proceedings of ESOCC 2013: Service-Oriented and Cloud Computing, 2nd European Conference, Málaga, Spain, September 11-13, 2013. LNCS 8135, pp. 1-3, Springer-Verlag, 2013.
DOI 10.1007/978-3-642-40651-5_1.

9. *Formal Aspects of Free and Open Source Software Components* with Roberto Di Cosmo, Ralf Treinen. In proceedings of FMCO 2012: HATS International School on Formal Models for Components and Objects, Bertinoro, Italy, 24-28 September 2012. LNCS 7866, pp. 216-239, Springer-Verlag, 2013.
DOI 10.1007/978-3-642-40615-7_8.

10. *Component Reconfiguration in the Presence of Conflicts* with Roberto Di Cosmo, Jacopo Mauro, Gianluigi Zavattaro. In proceedings of ICALP 2013: 40th International Colloquium on Automata, Languages and Programming, Riga, Latvia, 8-12 July, 2013. LNCS 7966, pp. 187-198, Springer-Verlag, 2013.
DOI 10.1007/978-3-642-39212-2_19.

11. *Why do software packages conflict?* with Cyrille Valentin Artho, Kuniyasu Suzaki, Roberto Di Cosmo, Ralf Treinen. In proceedings of MSR 2012: 9th IEEE Working Conference on Mining Software Repositories, co-located with ICSE 2012, IEEE, ISBN 978-1-4673-1760-3, pp. 141-150. June 2-3, Zurich, Switzerland.
DOI 10.1109/MSR.2012.6224274.

12. *Towards a Formal Component Model for the Cloud* with Roberto Di Cosmo, Gianluigi Zavattaro. In proceedings of SEFM 2012: 10th International Conference on Software Engineering and Formal Methods, Thessaloniki, Greece, 1-5 October, 2012. LNCS 7504, pp. 156-171, Springer-Verlag, 2012.
DOI 10.1007/978-3-642-33826-7_11.

13. *Learning from the Future of Component Repositories* with Pietro Abate, Roberto Di Cosmo, Ralf Treinen. In proceedings of CBSE 2012: 15th International ACM SIGSOFT Symposium on Component Based Software Engineering, Bertinoro, Italy, June 26-28,

2012. ISBN 978-1-4503-1345-2, pp. 51-60, ACM 2012. Award: Best Paper Award.
DOI 10.1145/2304736.2304747.

14. *MPM: a modular package manager* with Pietro Abate, Roberto Di Cosmo, Ralf Treinen. In proceedings of CBSE 2011: 14th International ACM SIGSOFT Symposium on Component Based Software Engineering, Boulder, Colorado, USA, 21-23 June, 2011. ISBN 978-1-4503-0723-9, pp. 179-188, ACM 2011. Award: ACM SIGSOFT Distinguished Paper Award
DOI 10.1145/2000229.2000255.
15. *Feature Diagrams as Package Dependencies* with Roberto Di Cosmo. In proceedings of SPLC 2010: 14th International Software Product Line Conference, Jeju Island, South Korea, 13-17 September 2010. LNCS 6287, ISBN 978-3-642-15578-9, pp. 476-480, Springer-Verlag, 2010.
DOI 10.1007/978-3-642-15579-6_40.
16. *The Ultimate Debian Database: Consolidating Bazaar Metadata for Quality Assurance and Data Mining* with Lucas Nussbaum. In proceedings of MSR 2010: 7th IEEE Working Conference on Mining Software Repositories, co-located with ICSE 2010, IEEE, ISBN 978-1-4244-6802-7, pp. 52-61. 02-03/05/2010, Cape Town, South Africa.
DOI 10.1109/MSR.2010.5463277.
17. *Content Cloaking: Preserving Privacy with Google Docs and other Web Applications* with Gabriele D'Angelo, Fabio Vitali. In proceedings of ACM SAC 2010: 25th Annual ACM Symposium on Applied Computing, ISBN 978-1-60558-639-7, pp. 826-830. 22-26/03/2010 - Sierre, Switzerland.
DOI 10.1145/1774088.1774259.
18. *Strong Dependencies between Software Components* with Pietro Abate, Jaap Boender, Roberto Di Cosmo. In proceedings of ESEM 2009: 3rd International Symposium on Empirical Software Engineering and Measurement, ISBN 978-1-4244-4842-5, pp. 89-99. October 15-16, 2009 - Lake Buena Vista, Florida, USA.
DOI 10.1109/ESEM.2009.5316017.
19. *A Model Driven Approach to Upgrade Package-Based Software Systems* with Antonio Cicchetti, Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio. In proceedings of ENASE 2009: 4th international conference on Evaluation of Novel Aspects to Software Engineering; held in conjunction with ICEIS 2009. 6-10 May 2009, Milan, Italy. CCIS Volume 69, pp. 262-276, Springer-Verlag, 2010.
DOI 10.1007/978-3-642-14819-4_19.
20. *Where are your Manners? Sharing Best Community Practices in the Web 2.0* with Angelo Di Iorio, Davide Rossi, Fabio Vitali. In proceedings of ACM SAC 2009: the 24th Annual ACM Symposium on Applied Computing. ISBN 978-1-60558-166-8, pp. 681-687, ACM.
DOI 10.1145/1529282.1529422.
21. *Wiki Content Templating* with Angelo Di Iorio, Fabio Vitali. In Proceedings of WWW 2008: 17th International World Wide Web Conference. April 21-25, 2008 Beijing, China. ACM ISBN 978-1-60558-085-2/08/04, pp. 615-624.
DOI 10.1145/1367497.1367581.
22. *Spurious Disambiguation Error Detection* with Claudio Sacerdoti Coen. In Proceedings of MKM 2007: The 6th International Conference on Mathematical Knowledge

Management. Hagenberg, Austria – 27-30 June 2007. LNAI 4573, Springer Berlin / Heidelberg, ISBN 978-3-540-73083-5, pp. 381-392, 2007.
DOI 10.1007/978-3-540-73086-6_30.

23. *Crafting a Proof Assistant* with Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi. In Proceedings of Types 2006: Types for Proofs and Programs. Nottingham, UK – April 18-21, 2006. LNCS 4502, Springer Berlin / Heidelberg, ISBN 978-3-540-74463-4, pp. 18-32, 2007.
DOI 10.1007/978-3-540-74464-1_2.

24. *From Notation to Semantics: There and Back Again* with Luca Padovani. In Proceedings of MKM 2006: The 5th International Conference on Mathematical Knowledge Management. Wokingham, UK – August 11-12, 2006. LNAI 4108, Springer Berlin / Heidelberg, ISBN 978-3-540-37104-5, pp. 194-207, 2006.
DOI 10.1007/11812289_16.

25. *A Content Based Mathematical Search Engine: Whelp* with Andrea Asperti, Ferruccio Guidi, Claudio Sacerdoti Coen, Enrico Tassi. In Proceedings of TYPES 2004: Types for Proofs and Programs. Paris, France – December 15-18, 2004. LNCS 3839, Springer Berlin / Heidelberg, ISBN 3-540-31428-8, pp. 17-32, 2006.
DOI 10.1007/11617990_2.

26. *A Generative Approach to the Implementation of Language Bindings for the Document Object Model* with Luca Padovani, Claudio Sacerdoti Coen. In Proceedings of GPCE'04 3rd International Conference on Generative Programming and Component Engineering. Vancouver, Canada – October 24-28, 2004 LNCS 3286, Springer Berlin / Heidelberg, ISBN 3-540-23580-9, pp. 469-487, 2004.
DOI 10.1007/b101929.

27. *Efficient Ambiguous Parsing of Mathematical Formulae* with Claudio Sacerdoti Coen. In Proceedings of MKM 2004: 3rd International Conference on Mathematical Knowledge Management. September 19-21, 2004 Bialowieza - Poland. LNCS 3119, Springer Berlin / Heidelberg, ISBN 3-540-23029-7, pp. 347-362, 2004.

Workshops internationaux, avec comité de lecture

1. *Using Preferences to Tame your Package Manager* with Pietro Abate, Roberto Di Cosmo, Louis Gesbert, Fabrice Le Fessant. In proceedings of OCaml 2014: The OCaml Users and Developers Workshop, September 5, 2014, Gothenburg, Sweden. Co-located with ICFP 2014. 2014.

2. *Sources of Inter-package Conflicts in Debian* with Cyrille Valentin Artho, Roberto Di Cosmo, Kuniyasu Suzaki. In proceedings of LoCoCo 2011 International Workshop on Logics for Component Configuration, affiliated with CP 2011

3. *Expressing Advanced User preferences in Component Installation* with Ralf Treinen. In proceedings of IWOCE 2009: International Workshop on Open Component Ecosystem, affiliated with ESEC/FSE 2009. Foundations of Software Engineering, ISBN 978-1-60558-677-9, pp. 31-40, ACM 2009.
DOI 10.1145/1595800.1595806.

4. *Towards maintainer script modernization in FOSS distributions* with Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio. In proceedings of IWOCE 2009: International Workshop on Open Component Ecosystem, affiliated with ESEC/FSE 2009. Foundations of Software Engineering, ISBN 978-1-60558-677-9, pp. 11-20, ACM 2009.
   DOI 10.1145/1595800.1595803.
5. *Package Upgrades in FOSS Distributions: Details and Challenges* with Roberto Di Cosmo, Paulo Trezentos. In proceedings of HotSWUp'08: Hot Topics in Software Upgrades. October 20, 2008, Nashville, Tennessee, USA. ACM ISBN 978-1-60558-304-4.
   DOI 10.1145/1490283.1490292.
6. *Streaming Validation of Schemata: the Lazy Typing Discipline* with Paolo Marinelli, Fabio Vitali. In Proceedings of Extreme Markup Languages 2007: The Markup Theory and Practice Conference. August 7-10, 2007 Montreal, Canada.
7. *Co-Constraint Validation in a Streaming Context* with Paolo Marinelli. In Proceedings of XML 2006: The world's oldest and biggest XML conference. Award: Winner of the XML Scholarship 2006 as best student paper. Boston, MA – December 5-7, 2006.
8. *Tinycals: Step by Step Tacticals* with Claudio Sacerdoti Coen, Enrico Tassi. In Proceedings of UITP 2006: User Interfaces for Theorem Provers. Seattle, WA – August 21, 2006. ENTCS (Elsevier, ISSN 1571-0661), Volume 174, Issue 2, pp. 125-142. May 2007.
   DOI 10.1016/j.entcs.2006.09.026.
9. *Constrained Wiki: an Oxymoron?* with Angelo Di Iorio. In Proceedings of WikiSym 2006: the 2006 International Symposium on Wikis. Odense, Denmark – August 21-23, 2006. ACM, 2006, ISBN 1-59593-417-0, pp. 89-98.
   DOI 10.1145/1149453.1149471.
10. *Searching Mathematics on the Web: State of the Art and Future Developments* with Andrea Asperti. In Proceedings of New Developments in Electronic Publishing AMS/SMM Special Session, Houston, May 2004 ECM4 Satellite Conference, Stockholm, June 2004 pp. 9-18. FIZ Karlsruhe, ISBN 3-88127-107-4.
11. *Brokers and Web-Services for Automatic Deduction: a Case Study* with Claudio Sacerdoti Coen. In Proceedings of Calculemus 2003: 11th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning. Roma, Italy – September 10-12, 2003, Aracne Editrice. ISBN 88-7999-545-6, pp. 43-57, 2003.

Revues nationales, avec comité de lecture

1. *Enforcing Type-Safe Linking using Inter-Package Relationships* with Mehdi Dogguy, Stéphane Glondu, Sylvain Le Gall. In Studia Informatica Universalis, Volume 9, Issue 1, pp. 129-157. Hermann 2011.

Conférences nationales, avec comité de lecture

1. *Enforcing Type-Safe Linking using Inter-Package Relationships* with Mehdi Dogguy, Stéphane Glondu, Sylvain Le Gall. In proceedings of JFLA 2010: 21st Journée Franco-

phones des Langages Applicatifs, pp. 29-54. 30/01-02/02/2010 - La Ciotat, France.

Rapports techniques

1. *Automatic Deployment of Software Components in the Cloud with the Aeolus Blender* with Roberto Di Cosmo, Antoine Eiche, Jacopo Mauro, Gianluigi Zavattaro, Jakub Zwolakowski. Inria technical report 2015.
2. *Optimal Provisioning in the Cloud* with Roberto Di Cosmo, Michael Lienhardt, Ralf Treinen, Jakub Zwolakowski. Aeolus project technical report, 7 Juin 2013.
3. *Component reconfiguration in the presence of conflicts* with Roberto Di Cosmo, Jacopo Mauro, Gianluigi Zavattaro. Aeolus project technical report, 22 Avril 2013.
4. *Common Upgradeability Description Format (CUDF) 2.0* with Ralf Treinen. Mancoosi project technical report 3, 24 November 2009.
5. *Strong Dependencies between Software Components* with Pietro Abate, Jaap Boender, Roberto Di Cosmo. Mancoosi project technical report 2, 22 May 2009.
6. *Metamodel for Describing System Structure and State* with Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio. Mancoosi project deliverable, D2.1, work package 2. January 2009.
7. *Description of the CUDF Format* with Ralf Treinen. Mancoosi project deliverable, D5.1, work package 5. November 2008.
8. *Stream Processing of XML Documents Made Easy with LALR(1) Parser Generators* with Luca Padovani. Technical report UBLCS-2007-23, September 2007, Department of Computer Science, University of Bologna.
9. *Templating Wiki Content for Fun and Profit* with Angelo Di Iorio, Fabio Vitali. Technical report UBLCS-2007-21, August 2007, Department of Computer Science, University of Bologna.

Thèses

1. *User Interaction Widgets for Interactive Theorem Proving* with . Ph.D. dissertation, Technical report UBLCS-2007-10, March 2007, Department of Computer Science, University of Bologna (advisor: Andrea Asperti; refereed by: Christoph Benzmueller, Marino Miculan).
2. *Web services per il supporto alla dimostrazione interattiva (Web services for interactive theorem proving)* with . Master thesis (Italian only), March 2003, Department of Computer Science, University of Bologna (advisor: Andrea Asperti; refereed by: Nadia Busi).

Part III

Selected Publications

CHAPTER 8

Dependency Solving: a Separate Concern in Component Evolution Management

This chapter contains the full text of the article "Dependency Solving: a Separate Concern in Component Evolution Management" *[6].*

Dependency Solving: a Separate Concern in Component Evolution Management☆

Pietro Abate[a], Roberto Di Cosmo[a,b], Ralf Treinen[a], Stefano Zacchiroli[a]

[a] *Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126, CNRS, F-75205 Paris, France*
[b] *INRIA Paris-Rocquencourt, F-75205 Paris, France*

Abstract

Maintenance of component-based software platforms often has to face rapid evolution of software components. Component *dependencies*, *conflicts*, and *package managers* with *dependency solving* capabilities are the key ingredients of prevalent software maintenance technologies that have been proposed to keep software installations synchronized with evolving component repositories.

We review state-of-the-art package managers and their ability to keep up with evolution at the current growth rate of popular component-based platforms, and conclude that their dependency solving abilities are not up to the task.

We show that the complexity of the underlying upgrade planning problem is NP-complete even for seemingly simple component models, and argue that the principal source of complexity lies in multiple available versions of components. We then discuss the need of expressive languages for user preferences, which makes the problem even more challenging.

We propose to establish dependency solving as a *separate concern* from other upgrade aspects, and present *CUDF* as a formalism to describe upgrade scenarios. By analyzing the result of an international dependency solving competition, we provide evidence that the proposed approach is viable.

Keywords: component, dependency solving, software evolution, package management, open source, competition

1. Introduction

A program that is used and that as an implementation of its specification reflects some other reality, undergoes continual change or becomes progressively less useful.

☆This work has been partially performed at IRILL http://www.irill.org

Email addresses: abate@pps.jussieu.fr (Pietro Abate), roberto@dicosmo.org (Roberto Di Cosmo), treinen@pps.jussieu.fr (Ralf Treinen), zack@pps.univ-paris-diderot.fr (Stefano Zacchiroli)

URL: http://www.pps.jussieu.fr/~abate (Pietro Abate), http://www.dicosmo.org (Roberto Di Cosmo), http://www.pps.jussieu.fr/~treinen/ (Ralf Treinen), http://upsilon.cc/~zack (Stefano Zacchiroli)

The above law of *Continuing Change* [20] applies to all evolving software systems, which are deemed to be the vast majority of existing systems [7]. The advent of Component-Based Software Engineering [6, 36] did not affect this fundamental truth: *mutatis mutandis* continuing change also holds for component-based systems [21]. The diffusion of rapidly evolving *component-intensive software platforms*—i.e. platforms where the number of components is in the tens or even hundreds of thousands—has raised the quality requirements for automatic tools that maintain component installations on behalf of users, be them developers, architects, administrators, or final users empowered to assemble components.

Component-intensive platforms are commonplace: FOSS (Free and Open Source Software) distributions (where components are called "packages"), development platforms like Eclipse and Apache Maven [9, 25] (which call components "plugins"), OSGi [29] ("bundles"), CMS communities ("add-ons"), Web browsers ("extensions"), and countless others. Despite apparent differences in terminology, all these platforms share concepts, properties, and problems. For instance, components have expectations on the deployment context: they may need other components to function properly—declaring this fact by means of *dependencies*—and may be incompatible with some other components—declaring this fact by means of *conflicts*. Those expectations must be respected not only at initial deployment-time, but also at each component release and for each individual component: a new version of a component cannot be deployed if its expectations are not met on the target system.

To maintain component assemblies, (semi-)automatic component manager applications are used to perform component installation, removal, and upgrades on target machines—we use the term *upgrade* to refer to any combination of those actions. Examples of component managers are as commonplace as component-intensive platforms: package managers, such as APT or Aptitude used in FOSS distributions to manage packages; P2 [19], used in Eclipse to deal with plugins; OSGi resolvers, which perform component deployment and configuration. These tools—called generically *package managers* in the following—incorporate numerous functionalities: trusted retrieval of components from remote repositories; planning of upgrade paths in fulfillment of deployment expectations (also known as *dependency solving*); user interaction to allow for interactive tuning of upgrade plans; and the actual deployment of upgrades by removing and adding components in the right order, aborting the operation if problems are encountered at deploy-time [10].

In contexts where the pace of component releases is rapid (e.g. FOSS [31, 14, 1]) the quality demand on package managers, and in particular on dependency solving, is very high. Package managers should: (1) devise upgrade plans that are correct (i.e. no plan that violates component expectations is proposed) and complete (i.e. every time a suitable plan exists, it can be found); (2) have performances that scale up gracefully at component repositories growth; (3) empower users to express preferences on the desired component configuration when several options exist, which is often the case. Surprisingly, all mainstream component manager applications the authors are aware of fail to address one or several of those concerns. Not addressing them is far from being a purely academic exercise, as Figures 1 and 2 show. Although anecdotal those and similar examples, which populate the experience of everyday package manager users, show that state-of-the-art component managers are short of fulfilling the aforementioned requirements.

```
# aptitude upgrade
1163 packages upgraded, 633 newly installed,
195 to remove and 0 not upgraded.
The following packages have unmet dependencies:
[...]
open: 4892; closed: 4995; defer: 170; conflict: 86
No solution found within the allotted time. Try harder? [Y/n]
Resolving dependencies...
open: 7592; closed: 7654; defer: 193; conflict: 89
open: 7798; closed: 7879; defer: 233; conflict: 89
open: 9938; closed: 9977; defer: 315; conflict: 89
No solution found within the allotted time. Try harder? [Y/n]
Resolving dependencies...
open: 14915; closed: 14952; defer: 372; conflict: 89
No solution found within the allotted time. Try harder? [Y/n]
Resolving dependencies...
open: 19880; closed: 19981; defer: 445; conflict: 89
No solution found within the allotted time. Try harder? [Y/n]
Resolving dependencies...
open: 25017; closed: 24998; defer: 467; conflict: 90
No solution found within the allotted time. Try harder? [Y/n]
Resolving dependencies...
open: 30110; closed: 29978; defer: 498; conflict: 91
No solution found within the allotted time. Try harder? [Y/n]n
```

Figure 1: Unexpected behaviour while using the legacy Aptitude package manager, on a FOSS system on the Debian GNU/Linux distribution. The user attempts to upgrade all components in need of upgrade on a machine equipped with the GNOME desktop environment and several LaTeX packages. The dependency solver loops and is unable to find a solution; after several attempts, the user gives up. (See http://bugs.debian.org/590470; retrieved November 29th, 2010.)

Considering the recent popularity of dependency-based abstractions in Component Based Software Engineering (CBSE, e.g. [17, 33, 11]), overlooking important dependency solving requirements appears to be dangerous.

This work provides substantial coverage of concepts and problems that are common in component managers equipped with automatic dependency solving abilities, for any non-trivial component model. Understanding such problems is of paramount importance because, in the context of component-intensive software platforms, software evolution is observed by users through the lens of component releases and often judged by the package manager abilities to successfully deploy new releases. Therefore, to avoid software evolution bottlenecks at the component deployment stage, we need to improve the ability of our tools to plan component upgrades. Unfortunately, as we will show, the problem is a hard one to tackle. In order to attack such a non-trivial and fairly overlooked problem, this paper proposes to treat dependency solving as a separate concern of component evolution and details the formalisms and technologies that are needed to enable such separation.

Paper contributions and structure. In Section 2 we present the upgrade planning problem, or simply *upgrade problem*, in a general setting, showing that in any non-trivial component model dependency solving is NP-complete. To tackle the problem, in Section 3 we propose to treat dependency solving as a separate concern, in order to share research and development efforts on upgrade planning. To that end, we need formalisms to: (1) capture upgrade scenarios coming from different component models in a unifying, well-defined semantics and (2) describe user preferences which are advanced enough to

```
# aptitude install baobab
[...]
The following packages are BROKEN: gnome-utils
The following NEW packages will be installed: baobab
[...]
The following actions will resolve these dependencies:
Remove the following packages:
  gnome gnome-desktop-environment libgdict-1.0-6
Install the following packages:
  libgnome-desktop-2 [2.22.3-2 (stable)]
Downgrade the following packages:
  gnome-utils [2.26.0-1 (now) -> 2.14.0-5 (oldstable)]
[...]
0 packages upgraded, 2 newly installed, 1 downgraded,
180 to remove and 2125 not upgraded. Need to get 2442kB
of archives. After unpacking 536MB will be freed.
Do you want to continue? [Y/n/?]
```

Figure 2: Attempt to install a disk space monitoring utility (called *baobab*) using Aptitude. In response to the request, the package manager proposes to downgrade the GNOME desktop environment all together to a very old version compared to what is currently installed. As shown in Section 6 a trivial alternative solution exists that minimizes system changes: remove a couple of dummy (or "meta") packages.

cover realistic use cases, but yet simple enough to be efficiently dealt with by state-of-the-art constraint solvers. Our proposals for those two formalisms are detailed in Sections 4 and 5. Section 6 validates the proposed approach by discussing an international dependency solving competition—called MISC—which has been run exploiting the proposed formalisms. Competition results show that state-of-the-art constraint solvers can easily outperform ad-hoc solvers embedded in mainstream package managers, confirming the thesis that separation of concerns and reuse are not only feasible, but also a viable strategy to improve upgrade planning and support component evolution.

2. Component Evolution and the Complexity of the Upgrade Problem

In this section we start by studying the complexity of the *upgrade problem* that package managers for component-intensive software platforms have to face. An important feature of the problem is that there is usually a multitude of possible choices. This has two consequences:

- For any given user request, there potentially exists an exponential number of solution candidates, which makes the problem NP-complete in all relevant cases (see Sections 2.1 and 2.2).
- There might be an exponential number of actual solutions to a problem instance, and we need a good way to pick the best among these solutions (see Section 2.3).

2.1. Complexity of the Upgrade Problem

A *software component* is a bundle of: (1) files that are to be installed on a target machine, (2) configuration logic to be executed at various stages of deployment, and (3) metadata which, among others, describe component expectations [10]. For the purpose of this paper we focus only on metadata since this is the information used by package managers to plan upgrades. There are different component models, but metadata contains at least the following features:

name: a component identifier that has a meaning over a time-line of releases;

version: an identifier of a specific release of a component that is meaningful relative to a given name;

dependencies: components that must be installed to make a component usable.

The expressiveness of the dependency language varies, but at the very minimum allows for a list of components that are required to be installed. More evolved models also allow for disjunctions (alternatives) and version constraints (like "component c in any version greater than 42"). Most component models also allow for:

conflicts: components that are not to be installed at the same time as the given component. Conflicts may come with version constraints, similar to dependencies.

features: names of virtual components *provided by* a component. They may be used to satisfy dependencies of other components and must not conflict with other installed components.

We assume that each package is uniquely identified by its name n and version v, and denote it as (n, v). A *repository* R is a set of components. An R*-installation* I is a set of components $I \subseteq R$ that has the properties of:

abundance: each package in I has its dependencies satisfied by packages in I;

peace: no package in I conflicts with another package in I.

The following theorem was proven, in a more specific context, in [12]:

Theorem 1. *Satisfiability of package upgrade requests is NP-complete, provided the component model features conflicts and disjunctive dependencies.*

Proof. First, we remark that the problem is clearly in NP since, given a subset of the repository, one can check in polynomial time that it satisfies abundance, peace, and the specific user request.

To prove NP-completeness, we show how to reduce the well known NP-complete problem 3-SAT to the upgrade problem. For this, we show that any instance of 3-SAT can be encoded into a simple instance of the upgrade problem, consisting of a single component installation request in an empty initial installation.

Let $F = C_1 \wedge \ldots \wedge C_n$ be an instance of the 3-SAT problem, where each C_i is a disjunction of three literals. We define a repository R_F that contains:

- for every literal L occurring in F a package $(L, 0)$ which conflicts with the package whose name is the complement of L,
- for every clause $C_i = L_i^1 \vee L_i^2 \vee L_i^3$ occurring in F a package $(C_i, 0)$ which depends on the disjunction of the packages $(L_i^1, 0)$, $(L_i^2, 0)$, and $(L_i^3, 0)$,
- a package $(F, 0)$ which depends on the conjunction of the packages $(C_1, 0), \ldots, (C_n, 0)$.

It is easy to see that F has a solution iff there is an R_F installation containing package $(F, 0)$. Note that no sophisticated usage of versions is needed for this encoding: we have used version 0 everywhere. □

The above proof makes essential use both of disjunctions in dependencies, and conflicts. In fact there are different ways how disjunctions in dependencies may appear: through explicit alternatives (as used in the proof), features, or multiple versions of a package. In fact, having both conflicts and disjunctions (in any form) are crucial for NP-completeness, as the following theorem shows:

Theorem 2. *Installability of a package in an empty environment is in PTIME in any of the two following cases:*

1. *The component model does not allow for conflicts.*
2. *The component model does not allow for disjunctive dependencies or features, and the repository does not contain multiple versions of packages.*

Proof. We first recall that component installability can be encoded into Boolean satisfiability [23]. Given a repository R, we construct a logical theory T_R as follows: we introduce, for every component or feature in R of name n and version v, a propositional variable X_n^v. A dependency d of package (n, v) is translated into an implication $X_n^v \to \bar{d}$, where $\bar{d}$ is the logical formula representing the dependency d, obtained by replacing an atomic dependency by the disjunction of all variables corresponding to components satisfying that atomic dependency[1]. For every conflict (n', v') of a package (n, v) we add a formula $\neg X_n^v \vee \neg X_{n'}^{v'}$. If packages $(n_1, v_1), \ldots, (n_k, v_k)$ provide feature f of version v then we add $X_f^v \to (X_{n_1}^{v_1} \vee \ldots \vee X_{n_k}^{v_k})$. It is easy to see that $T_R \wedge X_n^v$ has a propositional model iff there exists an R-installation that contains component (n, v). The formula $T_R \wedge X_n^v$ falls into particular classes in the two cases of the theorem:

1. If there are no alternatives in dependencies, no features, and no multiple versions of packages then all implications obtained from dependencies are of the form $X \to (X_1 \wedge \ldots \wedge X_n)$, which is equivalent to $(X \to X_1), \ldots, (X \to X_n)$. Since clauses obtained from conflicts are always binary, and since the formula X_n^v is unary, one obtains a theory which is a set of unary and binary clauses. The PTIME results follows since satisfiability of sets of unary and binary clauses is decidable in polynomial time [32].
2. If there are no conflicts then one just has formulas $(X_n^v \to \bar{d})$. Since all occurrences of literals in $\bar{d}$ are positive, we can rewrite each of these formulas by transforming $\bar{d}$ into disjunctive normal form as a set of clauses of the form $X_n^v \to (L_1 \vee \ldots \vee L_n)$. These are *dual Horn clauses*, that is clauses that contain at most one negative literal. Satisfiability of sets of dual Horn clauses is again decidable in PTIME ([32], who calls them *weakly positive clauses*).

□

2.2. Complexity in the Case of Component Evolution

The problem of package installation becomes significantly harder when one imposes that old versions of packages have to be *replaced* by new versions of packages, instead of just installing old and new version at the same time. This requirement appears in different form in different component models:

[1] This disjunction is empty, yielding the formula $\bot$, in case the package mentioned in the dependency is absent from the repository.

- The Debian package model allows to install only one version of a package at a time.
- In the RPM package model, it is a priori possible to install multiple versions of a package at a time; however it is in practice almost always excluded by the fact that different versions of a package install files with the same path on the file system, and hence are in conflict with each other.
- The Eclipse model allows for an explicit *singleton* property in component meta-data, with the semantics that only one version of that component must be installed.

In order to state a complexity result, we will consider in this subsection that component installations must be **flat**, that is must not contain two packages with the same name (which then would have different version). The complexity result stated in the following theorem, however, applies equally to the other models mentioned above since we may always require uniqueness of version for the packages used in the proof.

Theorem 3. *Existence of a* flat *installation containing a component is NP-complete, even when the component model does not allow for explicit conflicts, alternatives, and features.*

Proof. The problem is in NP for the same reason as in Theorem 1: one can check in polynomial time for every subset of the repository whether it satisfies abundance, peace, flatness, and the specific user request.

We show NP-completeness by giving a polynomial reduction of the 3-SAT problem. Let $F = C_1 \wedge \ldots \wedge C_n$ be a problem instance, where each C_i is of the form $C_i = L_i^1 \vee L_i^2 \vee L_i^3$. We define a repository R_F consisting of the following components:

- for each propositional variable X a package with name X, existing in versions 0 and 1. Each of these versions has no explicit conflicts or dependencies.
- for each clause C_i a package C_i in three versions 1, 2, and 3. None of them has conflicts. If the literal L_i^j $(j = 1, 2, 3)$ is a positive literal X then component (C_i, j) depends on $X(= 1)$. If the literal L_i^j is a negative literal $\neg X$ then component (C_i, j) depends on $X(= 0)$.
- a package of name F and version 1 that depends on $C_1, \ldots, C_n$.

If there is a flat R_F-installation containing $(F, 1)$ then F is satisfiable : Any flat installation may in particular contain at most one version of any package associated to a propositional variable. Hence, a flat R_F-installation I defines a propositional valuation α_I (if I does not contain any version of a package X then we may choose $\alpha_I(X)$ arbitrarily), and when I contains $(F, 1)$ then α_I obviously satisfied the 3-SAT instance F.

If F is satisfiable then there exists a flat R_F-installation containing $(F, 1)$: Let α be a solution of F. This means that one may choose, for any clause C_i, one index $s(i) \in \{1, 2, 3\}$ such that α satisfies the literal $L_i^{s(i)}$. We construct an R_F-installation from all the packages corresponding to propositional variables in the version according to their respective truth value in α, the packages $(L_1, s(1)), \ldots, (L_n, s(n))$, and finally the package $(F, 1)$. □

In some sense, a dependency on a package with name n acts like an exclusive choice in case of the flatness requirement on installations. If we have versions 1, 2 and 3 of packages with name n, then an unqualified dependency on name n can be read as the requirement on exactly one of $(n, 1)$, $(n, 2)$, $(n, 3)$.

For the problem to be NP-complete, it is enough to have just two versions of each component:

Corollary 1. *Theorem 3 holds for repositories containing at most two versions per package.*

Proof. It is sufficient to replace in the above proof each of the components C_i by two components, C_i^1 and C_i^2, each of them coming in version 1 and 2:

- $(C_i^1, 1)$ depends on (X, v) corresponding to the first literal of C_i,
- $(C_i^1, 2)$ depends on C_i^2,
- $(C_i^2, 1)$ depends on (X, v) corresponding to the second literal of C_i,
- $(C_i^2, 2)$ depends on (X, v) corresponding to the third literal of C_i,

The component $(F, 1)$ depends on $C_1^1, \ldots, C_n^1$. □

2.3. Dealing with exponentially many solutions

Having established the complexity of finding *a* solution to an upgrade problem, we now turn our attention to *the amount* of existing solutions for any given user request. The interest in analyzing that aspect stems from the observations that, among all possible solutions, package managers generally try to offer to the user the "best" solution, at least according to some predefined strategy. Indeed an often overlooked fact is that a user request that consists of just a list of components to install, remove or upgrade may have exponentially many solutions. This is closely related to the complexity results of the previous section which rely on the fact that there is an exponential number of solution *candidates*.

Example 1. *Consider a repository R consisting of components q_i, for $1 \leq i \leq n$, in versions 1 and 2, and a component p in version 1 depending on all of $q_1, \ldots, q_n$ in any version. The initial installation contains each of the package q_i in version 1, and we ask to install package p, where installations have to be flat.*

Any of the 2^n configurations $\{(p, 1)\} \cup \{(q_i, i) | i \in 1 \ldots n, 1 \leq i \leq 2\}$ is a solution.

These 2^n solutions are all pretty different from a user point of view. The solution that keeps the originally initially version of all the q_i may be preferred by "*paranoid*" users who want to avoid unnecessary changes to the system (as it is often the case for system administrators of critical production servers). The solution that changes all the q_i to their most recent version might be preferred by "*trendy*" users willing to have a system as up to date as possible (which is the case for many desktop and developer users).

State of the art package managers try to handle this issue by incorporating hard-wired criteria (most of which would give preference to the trendy solution above) and sometimes provide a bit of flexibility by means of cumbersome mechanisms that let the

user alter the standard solver behavior, like the *pinning* schema used by APT [28], or an API for programming custom criteria in Smart[2] and libzypp.[3]

Such ad-hoc mechanisms suffer from two main drawbacks: (1) they are package manager specific and therefore cannot be shared among different tools, preventing the development of common good practices in component deployment; (2) they are not expressive enough to encode all but the simplest use cases, making it difficult to precisely specify user needs. The right approach is—on the user side—to expose a high-level, solver independent, flexible mechanism to specify user preferences and—on the package manager side—to enable solver externalization and reuse.

3. Dependency solving as a separate concern

We have seen how dependency solving is a difficult, recurrent, and apparently underestimated problem. Re-developing from scratch dependency solvers as soon as dependencies and conflicts are introduced in yet another component model does not seem to have not served well users of component based systems. We argue that an alternative, more modular, approach is possible by treating *dependency solving as a separate concern* from other component management concerns. The goal is to decouple the evolution [sic] of dependency solving from that of specific package managers *and* component models.

We believe such a separation will benefit, at first, the involved scientific communities: CBSE and constraint solving. The former will gain the attention of the latter and will avoid to reinvent (solving) wheels, the latter will get access to a corpus of challenging upgrade problems to better tune existing solvers and techniques. Later on, we posit that synergies among the involved stakeholders will benefit final component users, by improving dependency solving abilities in state-of-the art package managers. Our early results seem to support these beliefs, as shown in Section 6.

To treat dependency solving as a separate concern, however, we need suitable abstractions and technologies that allow to describe upgrade problems in a way which is agnostic from specific component models and tools. In particular, we need ways to grasp all the information that describe any given upgrade problem instance:

1. *installed and available components*—describing all known components (local and remote) and information about which are currently installed;
2. *user request*—detailing the components that are requested to be installed, removed or upgraded, possibly with version constraints;
3. *user preferences*—the criteria describing how a user wants to choose a preferred solution out of the many possible ones.

In the following we present a Domain Specific Language (DSL)—called CUDF—able to encode (1.) and (2.), as well as a formalism defined on top of it to grasp (3.). Taken together they provide an unified way to capture all of the above in a unified way, which is both independent from component model details and rigorously defined to enable independent implementations of upgrade problem solvers. Having those devices available,

[2] `http://labix.org/smart`, retrieved December 2010
[3] `http://en.opensuse.org/Portal:Libzypp`, retrieved December 2010

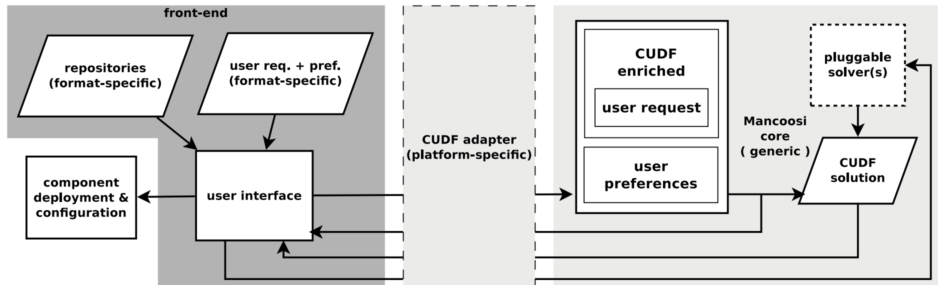


Figure 3: Modular package manager architecture

we can build adapters for each component platform and then build a modular solver engine where solvers can be plugged in according to user needs. Even more so, solvers can be run in parallel locally or outsourced (e.g. to solver farms in the "cloud"), in order to provide the user with the best solution current techniques and technologies can find.

The resulting modular architecture is shown in Figure 3. In such an architecture separation of concerns is established as following: *package manager* developers may focus on the killer features of their software (trust management, user interface and interaction, transactional upgrade deployment, etc.) and stop worrying about dependency solving issues; *CUDF adapters* are created for each component model and maintained by component metadata architects, or by CUDF experts working with them; *dependency solvers* are maintained by solver experts, who will see their technology gain many new fields of application by just supporting one generic I/O format—CUDF—which comes with a rigorous semantics, relieving the pain of interpreting the meaning of platform-specific component metadata.

4. A unified description of upgrades

To enable treating dependency solving as a separate concern in component upgrade planning, we need a language able to capture all relevant aspects of upgrade problem instances. In this section we present a DSL called CUDF (for Common Upgrade Description Format), whose documents describe instances of the component upgrade problem. The design of CUDF has been guided by a few general principles:

Platform independence. CUDF is a *common* format to describe upgrade scenarios coming from diverse environments. As a consequence, CUDF makes no assumptions on specific component model, version schema, dependency formalism, or package manager.

Solver independence. In contrast to encodings of inter-component relations which are targeted at specific solver techniques (see Section 7), CUDF stays close to the original problem, in order to preserve its structure and avoid bias towards specific solver.

Readability. CUDF is a compact plain text format which makes it easy for humans to read upgrade scenario, and ease interoperability with package managers.[4]

[4] As evidence of the benefits of this choice, CUDF is routinely used by the Eclipse P2 team to reason about upgrade scenarios, instead of the native XML encoding that comes with Eclipse. See http:

Extensibility. Only core component properties that are shared by mainstream platforms and essential to grasp the meaning of upgrade scenarios are predefined in CUDF. Other auxiliary properties can be declared and used in CUDF documents, to allow the preservation of relevant information that can then be used in optimization criteria, e.g. component size, number of bugs, etc.
Formal semantics. CUDF comes with a rigorous semantics that allows package manager and solver developers to agree on the meaning of upgrade scenarios. For example, the fact that self-conflicts are ignored is not a tacit convention implemented by some obscure line of code, but a property of the formal semantics.

4.1. Language overview

An upgrade scenario is represented by a *CUDF document*. It consists of a sequence of *stanzas*, each of which is a collection of key-value pairs called *properties*. Properties are typed within a simple *type system* containing basic data types (integers, booleans, strings) and more complex, component-specific data types such as boolean formulae over versioned components used to represent inter-component relationships.

Each CUDF document is made up of three logical sections: a *preamble*, a component *universe*, and a *request*. The universe contains one *component stanza* for each component known to the package manager, so both installed and non-installed (but available) components are represented uniformly in a document, in contrast to current platforms which often distribute this information in different locations using different formats. Component stanzas support a set of core properties (possibly optional, with default values), the most important of which are: `package` and `version` (which uniquely identify a component in the universe), `depends` and `conflicts` (context requirements), `provides` (*features* provided by the component), and `installed` (whether the component is installed).

Figure 4 shows a sample CUDF document. The component universe contains several component stanzas, where both core and extra properties are used. Extra properties must be declared in the *preamble*, which starts the document. Extra properties account for extensibility of the format and enable type checking of CUDF documents. A *request stanza* encodes the user request and concludes the document. In its general form, the request stanza details the components the user wants to `install`/`remove`/`upgrade` (using the homonymous properties), possibly specifying version requirements.

The full syntax of CUDF is given, as an EBNF grammar, in Appendix A; the formal semantics in Appendix B.

4.2. Expressiveness

As CUDF lays at the "interface" between package managers and dependency solvers, its expressiveness should be validated looking from both angles. From the point of view of package managers, we have shown that upgrade scenarios coming from several major component models can be encoded in CUDF; adapters are already available for: Debian and RPM packages,[5] Eclipse [5]—with an extension for full OSGi bundles in the working—, and common feature diagram formalisms used in software product lines [11].

`//wiki.eclipse.org/Equinox/p2/Meetings/20091221`, retrieved November 2010.

[5]both are supported out-of-the-box by Mancoosi tools [24]

```
preamble:
property: bugs: int = 0, suite: enum(stable,unstable) = "stable",

package: car
version: 1
depends: engine, wheel > 2, door, battery <= 13
installed: true
bugs: 183

package: bicycle
version: 7
suite: unstable

package: gasoline-engine
version: 1
depends: turbo
provides: engine
conflicts: engine, gasoline-engine
installed: true
...
request:
install: bicycle, gasoline-engine = 1
upgrade: door, wheel > 3
```

Figure 4: Sample CUDF document

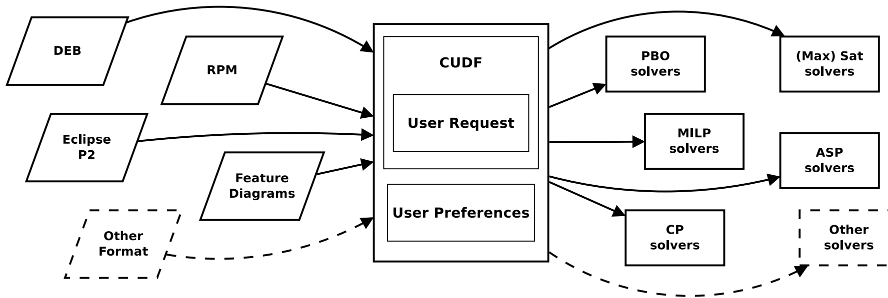


Figure 5: Sharing upgrade problems and solvers among communities

All encodings are *linear* in the number of components to encode, even in the presence of XOR dependencies.[6]

From the converse angle, that of dependency solvers, we observe that entrants in the MISC competition (see Section 6) have used very different solver technologies: boolean satisfiability (SAT), Mixed Integer Linear Programming (MILP), Answer Set Programming (ASP), and graph constraints. They have all been able to handle upgrade problems encoded as CUDF documents, providing convincing evidence that CUDF is adequate for a large spectrum of solving techniques.

Hence, at the time of writing, CUDF is already a unique pivot format that allows on one hand to share solvers among different package managers, and on the other hand to share a corpus of challenging upgrade problems among solver communities, as shown in Figure 5. The number of supported solver technologies and component frameworks has

[6]while usual SAT encodings blow up quadratically in the number of XOR dependencies.

grown steadily over the past years and it is likely to keep growing in the future.

4.3. Implementations

CUDF has been subject to an ad-hoc standardization process, resulting in a specification [37]. `libcudf` is the "reference" implementation of the specification; it consists of a parsing and pretty-printing library for CUDF, as well as an implementation of CUDF semantics. The latter consists in:

1. given a CUDF document, `libcudf` can verify whether installed components are consistent, i.e. whether they satisfy abundance and peace;
2. given a CUDF document and an encoding of a potential solution, `libcudf` can verify whether the solution is valid, i.e. abundance, peace, and request satisfaction.

`libcudf` comes with the `cudf-check` command line tool which provides the above two features out of the box. `libcudf` is Free Software and can be used both from the OCaml and C programming languages; it is available at http://www.mancoosi.org/software/.

The authors are aware of other CUDF implementations. Some have been developed in the context of the Mancoosi project to capture FOSS distribution upgrade scenario descriptions into CUDF, in order to build a cross-distribution corpus of upgrade problem instances [3]. Using the tools we have verified that the average size of an upgrade scenario encoded in CUDF is linear with the size of the original package manager information and usually smaller, since metadata not relevant for describing the upgrade problem can be dropped. For instance, on a large Debian installation, using both testing and unstable suites (totaling $\approx$ 45'000 packages), APT information on disk amounts to 14 Mb while the corresponding CUDF document is only 9 Mb.

An independent CUDF implementation is also available in CUPT,[7] a recent APT-compatible package manager for Debian. In CUPT, CUDF is used as an interface format to pipe upgrade scenarios to external solvers, so that upgrade planning can be decoupled from other package manager activities. While no stable software has been released yet, work is ongoing to implement CUDF in APT and APT2 in order to decouple dependency solving from the package managers.

5. User preferences as multicriteria optimization

The DSL presented in the previous section addresses the need of grasping those aspects of an upgrade scenario that are related to the *correctness* of a given solution (i.e. "does the solution satisfy the user request as well as the expectations of all installed components?"). *Quality* aspects of solutions (i.e. "is the proposed solution to my liking?") are much less known, not to mention agreed upon, and hence they do not yet constitute suitable material for DSL standardization. Nonetheless, to improve the state-of-the-art in upgrade planning we do need at the very least a rigorous framework to reason about solution quality. In this section we propose one such formalism.

As we have seen in Section 2, there are in general exponentially many solutions to a user request, so it is necessary to allow users to express their preferences about the

[7] http://wiki.debian.org/Cupt, retrieved December 2010

desired solution. The state-of-the-art approach is to present one particular solution—found according to some built-in strategies generally unknown to the user—and then allow the user to interactively fiddle with the solution. Considering again Example 1, it is easy to see why this approach has serious shortcomings: a "paranoid" user who is presented with a "trendy" solution will need to make n changes to the solution (and usually rerun the solver each time) before getting what she wants. In modern component repositories, where n can be quite large, this approach is not viable.

An alternative approach is to let the user specify high-level criteria that capture what she considers important in a solution: she may be concerned about the packages that are *changed*, the packages that are *not up to date*, the packages that get *removed*, or even "the number of installed security fixes", or "the overall installed size". On top of CUDF semantics, we can build an extensible dictionary of well-defined criteria like the above and then let the user inform the solvers that the required solution should maximize, or minimize, a given criterion.

It is quite natural for the user to combine several of these criteria: to compare two solutions s and s' whose criteria have values $(c_1, \ldots, c_n)$ and $(c'_1, \ldots, c'_n)$, the user will prefer s over s' if all criteria of s are better or equal than s' (i.e. s is Pareto-better then s'). Unfortunately, when one has more than one criterion, there may be many incomparable Pareto-optimal solutions; this is the core problem of *multicriteria optimization* which has been extensively studied in the optimization research community [34]. Many different approaches have been proposed to aggregate multiple criteria, the most common being:

Lexicographic. The criteria are ordered by importance, and compared lexicographically: $(c_1, \ldots, c_n)$ is better than $(c'_1, \ldots, c'_n)$ iff there exists a i s.t. for all $j < i$ $c_j = c'_j$, and $c_i > c'_i$; for example, a security upgrade may be considered more important than any other criterion, and put first in the order.

Weighted sum. The criteria are aggregated into a single measure using user-specified *weights* k_i: $(c_1, \ldots, c_n)$ is better than $(c'_1, \ldots, c'_n)$ iff $\sum_{1 \leq i \leq n} k_i c_i > \sum_{1 \leq i \leq n} k_i c'_i$; this may be useful when trying to balance different criteria for which no clear order is established.

More sophisticated approaches exists, like *leximin* and *leximax* [13], and an extensive literature is devoted to them. According to the use case, the best aggregation function may vary widely. Our own proposal for a high level user preferences formalism is simple yet expressive:

1. define a dictionary of useful criteria c_i;
2. define a dictionary of aggregation functions *lex*, *weightedsum*, *leximin*, etc.
3. write the user preference as an expression $op(k_1 c_1, \ldots, k_n c_n)$ where k_i can be one of $\{+, -\}$ to indicate maximization or minimization of the criterion (for aggregation functions like *lex*, *leximin* and *leximax*), or an integer (for aggregation functions like *weightedsum*).

Formally we define the criteria as in Table 1, where I is the initial installation and S is a proposed new installation. We write $V(X, name)$ the set of versions in which *name* (the name of a component) is installed in X, where X may be I or S. That set may be empty (*name* is not installed), contain one element (*name* is installed in exactly that version), or even contain multiple elements in case a component is installed in multiple

Table 1: Optimization criteria

$removed(I, S)$	$=\{name \mid V(I, name) \neq \emptyset$ and $V(S, name) = \emptyset\}$
$new(I, S)$	$=\{name \mid V(I, name) = \emptyset$ and $V(S, name) \neq \emptyset\}$
$changed(I, S)$	$=\{name \mid V(I, name) \neq V(S, name)\}$
$notuptodate(I, S)$	$=\{name \mid V(S, name) \neq \emptyset$ and does not contain the most recent version of name in $S\}$
$unsatrec(I, S)$	$=\{(name, v, c)$ — v is an element of $V(S, name)$ and $(name, v)$ recommends $..., c, ...$ and c is not satisfied by $S\}$

versions.[8] Using this formalism, it is quite easy to define a *paranoid* preference as

$$paranoid = lex(-removed, -changed)$$

The solution scoring best under this criterion will be the one with the minimum number of removed functionalities, and then with the minimum number of changes. A *trendy* preference is also easy to write

$$trendy = lex(-removed, -notuptodate, -unsatrec, -new)$$

Currently, each criterion and aggregation function must be specifically encoded for a given solver technology, but work on a generic system which will be able to produce these encodings automatically is ongoing [38].

6. Experimental results: the Mancoosi International Solver Competition

The DSL and formalism presented in the previous two sections have been used to run a dependency solving competition called MISC, for Mancoosi International Solver Competition. The variety of solving techniques implemented by participants, as well as the popularity of FOSS distributions from which package manager entrants come, give, in the authors opinion, a reasonable guarantee of the generality of the following results, that come from MISC 2010, the first edition of the competition:

1. The proposed languages and formalisms are expressive enough to encode both *real* upgrade scenarios coming from users of popular FOSS distributions and *synthetic* problems of increasing complexity.
2. The proposed languages and formalisms are unbiased enough to allow constraint solvers, based on a wide range of techniques, to attack upgrade problem instances. Dependency solving can therefore be outsourced to external solvers, as depicted in Figure 3.
3. The complexity of real upgrade problem instances grows with the number of component repositories, as well as the complexity of the optimization criteria.

[8]The CUDF component model is not flat but allows to encode both flat and non-flat models [3].

4. Dependency solving abilities of package managers used in popular FOSS distributions fall short of state-of-the-art constraint solvers, both in terms of solution quality and completeness.

In this section we present and discuss MISC 2010 results, as evidence of the above claims.

6.1. Competition details

MISC 2010 has been run in June 2010. Its results have been presented at the LoCoCo workshop, in the context of FLoC (Federated Logic Conference) 2010. All competition data (formal rules, problem instances, results, etc.) are available at http://www.mancoosi.org/misc-2010/ and allow to independently re-run the competition.

Each participant had to face several problem instances. For each instance, the solver received a full CUDF document as input and must produce a CUDF-encoded solution (i.e. a CUDF document without a request stanza). Solvers could participate in either one or both of two tracks—trendy and paranoid, as defined in Section 5—and strove to optimize their solutions accordingly. Problem instances were classified in categories: synthetic problems (categories: *easy*, *difficult*, *impossible*), instances of the problem 1-in-3 SAT (category *cudf_set*), and real instances collected from Debian users (category *debian-dudf*) using the `mancoosi-contest` utility [24] which plugs into package managers for Debian-based distributions to store upgrade problems in CUDF format.

Synthetic problems have been generated from a real Debian installation by varying a number of parameters such as the number of components in the universe, the number of installed components, and the number of components requested to install/remove/upgrade (i.e. request size). The size of requests ranges from 10 (for *easy*) to 20 (*impossible*) and components appearing therein are possibly equipped with version constraints.

For the *difficult* and *impossible* categories, the initial state has been made on purpose inconsistent by marking random components as installed, ensuring their context requirements were not satisfied, to simulate a badly broken user installation.

The following solvers took part in the competition:

solver	author/affiliation	technique/solver
apt-pbo [39]	Trezentos / Caixa Magica	Pseudo Boolean Optimization
aspcud	Matheis / University of Potsdam	Answer Set Programming
inesc [4]	Lynce et. al / INESC-ID	Max-SAT
p2cudf [4]	Le Berre and Rapicault / Univ. Artois	Pseudo Boolean Optimization / Sat4j (www.sat4j.org)
unsa [26]	Michel et. al / Univ. Sophia-Antipolis	Mixed Integer Linear Programming / CPLEX (www.cplex.com)

No solver has been provided by the authors, who acted solely as competition organizers. We added two extra participants—apt-get and aptitude—by wrapping with a CUDF-compatibility layer the solvers of package managers used in Debian-based FOSS distributions. As they do not allow to specify preferences, the purpose of the experiment was to check how hard-coded optimizations score with respect to competition criteria. The solver *ucl* from Gutierrez et. al from Univ. Louvain that took part to the Misc competition it is not presented here as its results were not relevant.

Table 2: MISC 2010 results: paranoid (above) and trendy (below) criteria. Each column show (before the parenthesis) the penalties accumulated by a participant, category by category: the lower, the better. Between parentheses is shown the aggregate solving time, in seconds, of a participant on all the problems of a given category: the lower the better, although time is used only to break ties between participants who accumulated the same amount of penalties. Highlighted cells, one per row, denote the winning participant of a given category.

Category	apt-pbo	aspcud	inescp	p2cudf	uns
cudf_set	162 (2700.00)	118 (1572.82)	108 (11.64)	108 (15.22)	111 (1984.21)
debian-dudf	236 (3673.76)	222 (3700.45)	32 (271.39)	32 (180.01)	86 (1336.49)
difficult	522 (1039.71)	58 (1553.82)	92 (298.99)	99 (3209.55)	55 (34.97)
easy	504 (177.10)	21 (408.48)	63 (122.44)	63 (121.97)	21 (18.59)
impossible	300 (3873.00)	270 (4500.00)	120 (1890.13)	120 (1924.79)	15 (151.27)
Total	1724 (11463.58)	689 (11735.57)	415 (2594.58)	422 (5451.54)	288 (3525.52)

Category	apt-pbo	aspcud	inesct	p2cudf	uns
cudf_set	135 (2700.00)	93 (2007.25)	90 (12.38)	90 (17.97)	107 (2332.43)
debian-dudf	211 (3743.04)	189 (3881.19)	194 (3711.60)	39 (5151.68)	74 (1782.70)
difficult	415 (1959.87)	70 (4439.18)	101 (1681.03)	98 (5223.88)	49 (83.80)
easy	410 (222.92)	46 (1034.32)	64 (317.52)	61 (597.80)	21 (24.85)
impossible	225 (4500.00)	190 (2671.69)	220 (4221.98)	181 (4182.90)	15 (734.58)
Total	1396 (13125.83)	588 (14033.63)	669 (9944.52)	469 (15174.24)	266 (4958.36)

Table 3: Solver comparison with a growing number of component repositories; paranoid (above) and trendy (below) criteria. Categories are identified by the initial(s) of Debian repositories (or *suites*), in the following order: `sarge`, `etch`, `lenny`, `squeeze`, `sid`. Cell values are to be interpreted in the same way as Table 2.

Category	apt-get	aptitude	aspcud	inescp	p2cudf	uns
s-e-l-s-s	176 (2435.49)	252 (1672.11)	210 (3000.00)	77 (1070.93)	24 (2861.72)	10 (78.13)
s-e-l-s	176 (2434.13)	231 (2331.87)	210 (3000.00)	58 (818.82)	23 (2855.84)	10 (63.90)
s-e-l	210 (3000.00)	245 (1621.60)	10 (389.68)	48 (443.98)	32 (1757.80)	10 (30.23)
s-e	194 (2706.06)	280 (116.07)	10 (166.57)	30 (84.17)	30 (555.35)	10 (14.64)
s	82 (918.40)	57 (33.80)	10 (68.14)	47 (31.87)	47 (268.22)	10 (3.68)
Total	838 (11494.07)	1065 (5775.45)	450 (6624.40)	260 (2449.77)	156 (8298.92)	50 (190.60)

Category	apt-get	aptitude	aspcud	inesct	p2cudf	uns
s-e-l-s-s	150 (2435.50)	198 (2419.24)	174 (2731.37)	180 (3000.00)	36 (2882.61)	10 (401.82)
s-e-l-s	150 (2433.72)	192 (2617.81)	180 (3000.00)	180 (3000.00)	20 (2866.66)	10 (318.83)
s-e-l	180 (3000.00)	204 (2000.62)	20 (2013.05)	50 (1949.77)	34 (2854.06)	10 (121.50)
s-e	166 (2706.06)	240 (208.88)	112 (2700.21)	26 (422.95)	31 (2281.39)	10 (34.35)
s	88 (918.37)	61 (34.55)	92 (2259.55)	34 (57.33)	34 (1158.47)	10 (5.32)
Total	734 (11493.65)	895 (7281.09)	578 (12704.17)	470 (8430.05)	155 (12043.18)	50 (881.82)

MISC 2010 results are given in Table 2. The clear winner in both tracks is *unsa*, followed by *p2cudf* for the trendy track and *inesc* for the paranoid track. It is important to notice that the solvers perform differently on different problem sets: for example, *p2cudf* shows better results than the others in the category *debian-dudf*, and it will be surely interesting to analyse, in future work, the structural differences among the different problem sets.

6.2. Discussion

CUDF acceptance. The actual run has been preceded by a discussion period among organizers and participants. During this period, solver authors could expose their doubts about CUDF semantics and competition rules, as well as submit solver prototypes for the only purpose of testing their CUDF-based interface with the competition infrastructure.

Solver authors have not reported any perceived bias, of either CUDF or the optimization criteria, towards specific solving techniques. The acceptance of the proposed languages and formalisms among participants has hence been very good, although self-selection bias is possible. The main discussion topics revolved around parsing issues and misconceptions about how upgrades “should” work. Interestingly, while CUDF semantics is rigorous and has proven to be very stable thus far, solver authors used to specific component platforms tend to believe upgrade should work as they are “used to”, even if the semantics of upgrades in their platform of origin (e.g. RPM) is ambiguous and delegated to implementation details of specific package managers. This aspect has reinforced our conviction that an interface format equipped with a rigorous semantics is the way to go in order to drive the attention of constraint solving communities to upgrade problem issues.

How complexity grows in practice. MISC 2010 results clearly show that the number of criteria in the optimization function is an important source of complexity: the trendy and paranoid tracks are run on the same problem sets, whereas trendy (which has more parameters than paranoid) is consistently more difficult to handle for all solvers.

We have also run all competition entrants on a separate set of problems, specifically designed to test the impact of having several repositories available, a scenario that happens quite often in practice. The corresponding categories have been built as follows: in a base universe, corresponding to the Debian distribution *sarge* (currently also known as *oldstable*), a fixed request and a fixed set of installed components is generated, ensuring it is satisfiable (meaning that the request has at least a solution, independently of the optimization criteria). The very same upgrade problem is then replicated in larger universes, by adding more recent Debian repositories: *etch*,*lenny*, *squeeze*, and *sid*.

Table 3 show the performance of the solvers on these categories. It is immediate to notice how the time needed to answer the same request grows very quickly when increasing the number of available component repositories. This is explained by the fact that using multiple repositories greatly increases the number of components available in multiple versions and, in turn, the number of conflicts in a flat component model.

Performance of FOSS package managers. Table 3 permits to assess the relative performances of competition entrants and package managers from popular FOSS distributions. Package manager solvers exhibit decent performances on machines equipped with a single component repository, which is often the case for freshly installed machines.

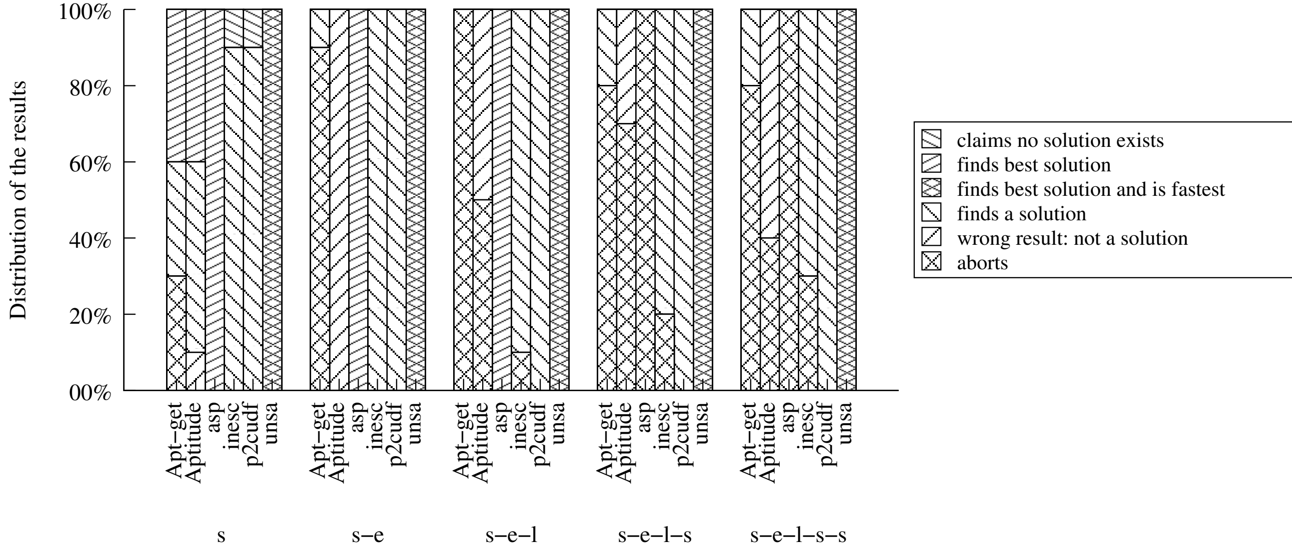


Figure 6: Solver results with increasing number of repositories (trendy)

This matches end-user experience that package manager performance on newly installed machines is quite good.

However, problems on machines that were installed from an old distribution and that use a mixture of component repositories, turn out to be very challenging. In Figure 6 we see clearly that, while *apt-get* and *aptitude* behave well with one repository, they become unreliable from two repositories on, and are no longer able to solve the large majority of problems at all. This corresponds to the end-user experience that installation and upgrades become less reliable on FOSS machines after a year or so: this corresponds more or less to the release cycle of several mainstream distributions, and end-users find themselves on machines where the package manager needs to handle more than one repository, the original one from which the machine was installed, and the newly released one.

Looking at the time distribution of the same experimental data in Figure 7 (for the trendy criterion) we observe a similar pattern. While state-of-the-art package managers abort or time-out, solvers like *unsa* or *inesc* are still able to cope with complex problems in less than 60 seconds. It is important to notice that in this context, a solution is "optimal" only with respect to solutions given by other solvers. Therefore even if a solver does not provide the best solution, it is still important to take it into consideration in the overall evaluation.

The data for the paranoid criterion in Table 3 shows the same overall behaviour, with significantly shorter execution times, indicating that the number of combined criteria in the user preferences is another significant factor in the complexity of the upgrade problems (paranoid involves 2 criteria, while trendy involves 4).

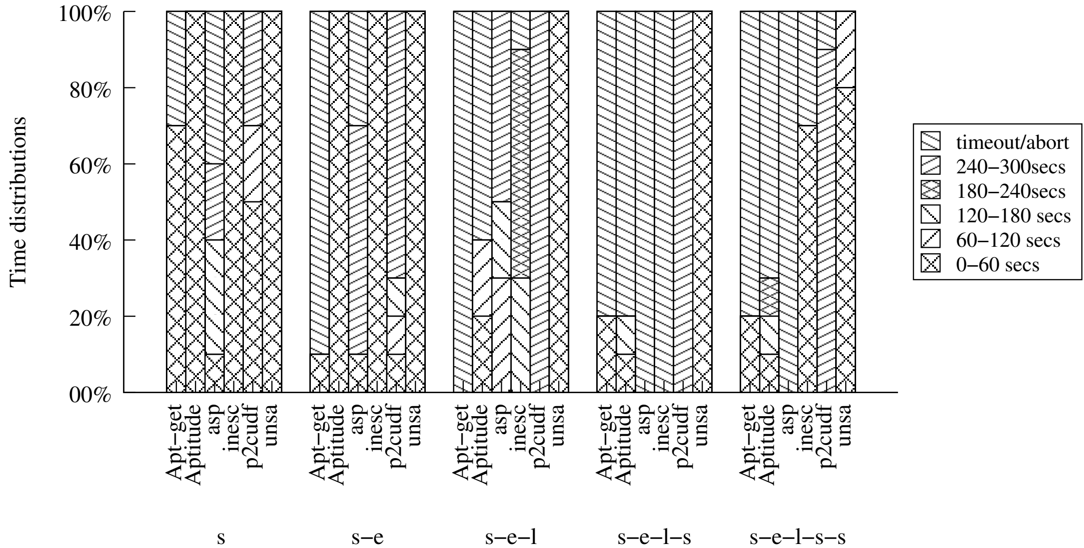


Figure 7: Solver performances with increasing number of repositories (trendy)

7. Related work

Software evolution management has many facets; in this paper we have focused on the area of post-development management. In particular we have studied how to improve software upgrade planning in package managers which are equipped with automatic dependency solving, given that such utilities are common place in component-intensive software platforms. The problem of dealing with inter-component relationships was known well-before the advent of such package managers, though. Seminal work in the area of software configuration management (e.g. [8, 27]) has established the "provide/require" paradigm to reason about component interconnection, with a varying degree of granularity and expressiveness [30]. Those and subsequent works have also detailed formal properties able to grasp, and practically verify, the compatibility of (new versions of) components within a given deployment context.

The explicit notion of inter-component conflicts is not part of those seminal proposals. In the technology camp such a notion has been popularised by the advent of early component managers (e.g. the FreeBSD porting system [35], RPM, and dpkg). Together with conflicts, early package managers have brought to users the folklore problem known as "dependency hell", i.e. the difficulty of satisfying at the same time all component dependencies and conflicts, sometimes stumbling upon the (apparent) impossibility of doing so. The next technology leap has brought package managers equipped with automatic dependency solving abilities (e.g. APT [28], Yum, Urpmi, etc.). Such systems have not only solved most of the issues brought by the dependency hell, but also addressed several of the concerns related to software distribution (see [16] for an introduction on the subject), even though they have done so in a centralized rather than federated way [15]. Where the state of the art in package managers is still lacking though, as we have shown,

is in their actual dependency solving ability. Improvement in that area is badly needed to properly plan upgrades in component-intensive software platforms.

Turning to formal encodings of the upgrade problem, an early SAT-encoding and complexity analysis for the upgrade problem, limited to the component models of FOSS distributions, has been provided by some of the authors in [12, 23] and has popularised the use of SAT technology in package managers. Results and encoding detailed in the present paper are more general and detail the minimum requirements for any component model to exhibit similar complexity behaviors.

The OPIUM prototype used in 2006 a SAT solver with an ad-hoc, hard coded optimization in line with the paranoid criterion [40]; SUSE's libzypp incorporated a SAT solver in 2007; the Eclipse P2 system comes with the Sat4J solver since 2007 [18]. This trend seems to continue steadily: a very recent entrant is apt-pbo, introduced in the Caixa Mágica distribution in early 2010 [39]. None of those systems have offered the ability to outsource dependency solving and optimization to an external solver.

The language we have proposed to encode user preferences is more flexible than those of OPIUM and similar experiences: we provide a core ontology of criteria and combinators to join them together. Even though user criteria must currently be specifically encoded for any given solver, we have looked at ways to automate the encoding. Moreover instead of leaving to the user the task of defining specific criteria, they could be asked for high level preferences and then use a goal-based model to "compile" those desiderata to the target criterion language. The work of Liaskos et al. [22], even if not directly related to our domain, goes in the direction of making complex systems easily configurable by deducing low-level options from high-level user specifications.

Several alternative encodings of the upgrade problem have been proposed: SAT [23, 40, 18], Pseudo Boolean Optimization [39], Partial Weighted Max SAT [4], Mixed Integer Linear Programming [26], as well as some others championed by entrants in the MISC 2010 competition (see Section 6).

Jenson [17] proposes a component model without explicit (or implicit) component conflicts and does not handle component removal in neither requests nor solutions. As a consequence, such a degenerate upgrade problem is way simpler than what we have modeled in this paper and can be solved in polynomial time, even though the number of solutions may be huge. Dependency solving as SAT with optimization has been reviewed in [18] where it was also observed that much of the complexity stems from multiple versions of components and the constraints they entail.

The need of dealing properly with dependencies in CBSE have been observed before [21, 41]. Vieira et. al have argued that dependencies should be treated as a first class problem in CBSE [41] and have established requirements for that. While we focus on static deploy-time dependencies, which have become popular in the meantime, we observe that CUDF fulfills all their requirements of "*being based on uniform design principles following some kind of standardization*" and offer dependency metadata which are "*expressive, intuitive, and concise* [in] *representation*". We agree with the authors and believe that the proposed formalisms are a significant step forward in treating dependencies as a first class problem in CBSE.

8. Conclusions

Dependency solving is difficult. This is hardly a surprise for anyone maintaining software installations, especially when they are made of thousands of components evolving rapidly and independently. The phenomenon requires nevertheless a detailed analysis to pinpoint the origin of the complexity. We found that, for common component models and platforms, the complexity is due to inter-component conflicts, either explicitly declared as component metadata or implicitly assumed between different versions of the same components. This theoretical result is confirmed by experimentation on both real and synthetic upgrade problem instances: dependency solving becomes harder as component repositories are added, thus increasing the number of available versions of the same components. Complexity also increases with the complexity of user preferences (i.e. optimization criteria). This explains why shortcomings of state-of-the-art dependency solvers are often not observed on freshly installed machines, but pop up as soon as one tries to do upgrades among distribution major releases, or else to mix and match components from different releases.

Better tools to support evolution of component-based systems are needed. Design, development, integration, and deployment of these new tools will only be made possible if we treat dependency solving as a separate concern of evolution management, i.e. as a first class research problem in its own right. To that end, we need rigorous abstractions to be put at the interface between component managers and solvers engineered by independent research communities, which enjoy the challenges posed by concrete upgrade problems. We have introduced some of those abstractions—CUDF and a companion user preference language—and we have reported the results of MISC, an international solver competition based on these abstraction, which confirm their adequateness.

On top of the proposed abstractions, it is easy to imagine a generic component manager front-end, which implements the architecture of Figure 3 and can then be targeted, adding back-ends, to specific component platforms. We have developed one such prototype, called MPM [2], targeting Debian-based FOSS distributions. Any solver implementing the interface of MISC 2010 can be plugged into MPM and used to plan package upgrades; upgrade deployment will then be delegated to legacy distribution tools. As an example, a trivial solution to the upgrade problem discussed in Figure 2 can be found by MPM using the *inesc* solver submitted to the paranoid track:

```
remove: gnome-utils gnome-desktop-environment gnome
install: baobab=2.4.2-1.1+b1
```

in such a solution the (virtual) packages `gnome-utils`, `gnome-desktop-environment`, and `gnome` are still removed, whereas all other packages forming the GNOME desktop are not, saving the (possibly newbie) user from losing her user-friendly work environment. This is a consequence of the paranoid criterion and of a dependency solver able to find a high-quality solution with respect to such desiderata.

As this example shows, one-size–fits-all solvers are not the way to go, especially when solvers are developed in house without reusing existing knowledge and results. Rather, we need highly customizable upgrade planners able to satisfy diverse user needs. Decoupling solvers from package managers is a necessary intermediate step that makes it possible to experiment with independent solvers, and to outsource dependency solving to evolution planners living far away from component managers.

As evidence of the pertinence of our approach, the experimental version `0.8.16 exp5` of `apt`, a mainstream package manager for the Debian distribution, implements a CUDF interface to call the solvers issued from the MISC competition.

Acknowledgments

The authors are grateful to the members of the Mancoosi and Eclipse P2 projects, for many stimulating discussions on the CUDF format, optimization criteria, and their practical use cases. A special acknowledgement goes to the optimization and solving communities who took part in the editions of the MISC competition: their solvers are the ingredients that make separation of concerns around dependency solving a concrete and useful approach.

References

[1] Abate, P., Boender, J., Di Cosmo, R., Zacchiroli, S., 2009. Strong dependencies between software components, in: ESEM 2009: International Symposium on Empirical Software Engineering and Measurement, IEEE. pp. 89–99.

[2] Abate, P., di Cosmo, R., Treinen, R., Zacchiroli, S., 2011. Mpm: A modular package manager, in: CBSE 2011, ACM.

[3] Abate, P., Guerreiro, A., Laurière, S., Treinen, R., Zacchiroli, S., 2010. Extension of an existing package manager to produce traces of ugradeability problems in CUDF format. Mancoosi deliv. D5.2. http://www.mancoosi.org/reports/d5.2.pdf.

[4] Argelich, J., Le Berre, D., Lynce, I., Marques-Silva, J., Rapicault, P., 2010. Solving Linux upgradeability problems using boolean optimization, in: LoCoCo: Logics for Component Configuration, pp. 11–22.

[5] Bozman, C., 2010. Converting Eclipse metadata into CUDF. Technical report 5. Mancoosi project. http://www.mancoosi.org/reports/p2.pdf.

[6] Brown, A.W., Wallnau, K.C., 1998. The current state of CBSE. IEEE Software 15, 37–46.

[7] Cook, S., Harrison, R., Lehman, M.M., Wernick, P., 2006. Evolution in software systems: foundations of the SPE classification scheme. Journal of Software Maintenance and Evolution 18, 1–35.

[8] DeRemer, F., Kron, H., 1975. Programming-in-the large versus programming-in-the-small. SIGPLAN Notice 10, 114–121.

[9] Des Rivières, J., Wiegand, J., 2004. Eclipse: a platform for integrating development tools. IBM Systems 43, 371–383.

[10] Di Cosmo, R., Trezentos, P., Zacchiroli, S., 2008. Package upgrades in FOSS distributions: Details and challenges, in: HotSWUp'08: International Workshop on Hot Topics in Software Upgrades, ACM. pp. 7:1–7:5.

[11] Di Cosmo, R., Zacchiroli, S., 2010. Feature diagrams as package dependencies, in: SPLC 2010: Software Product Lines Conference, pp. 476–480.

[12] EDOS Project, 2006. Report on Formal Management of Software Dependencies. EDOS Project Deliverables D2.1 and D2.2.

[13] Fishburn, P.C., 1974. Lexicographic orders, utilities and decision rules: A survey. Management Science 20, 1442–1471.

[14] Gonzalez-Barahona, J., Robles, G., Michlmayr, M., Amor, J., German, D., 2009. Macro-level software evolution: a case study of a large software compilation. Empirical Software Engineering 14, 262–285.

[15] Hall, R.S., Heimbigner, D., van der Hoek, A., Wolf, A.L., 1997. An architecture for post-development configuration management in a wide-area network, in: 17th International Conference on Distributed Computing Systems, IEEE. pp. 269–278.

[16] van der Hoek, A., Hall, R., Heimbigner, D., Wolf, A., 1997. Software release management, in: Software Engineering — ESEC/FSE'97. Springer Berlin / Heidelberg. volume 1301 of *LNCS*, pp. 159–175.

[17] Jenson, G., Dietrich, J., Guesgen, H., 2010. An empirical study of the component dependency resolution search space, in: CBSE 2011: International ACM Sigsoft Symposium on Component Based Software Engineering, Springer. pp. 182–199.

[18] Le Berre, D., Parrain, A., 2008. On SAT technologies for dependency management and beyond, in: SPLC 2008: Software Product Lines Conference, 2nd Volume, pp. 197–200.

[19] Le Berre, D., Rapicault, P., 2009. Dependency management for the Eclipse ecosystem, in: IWOCE 2009: International Workshop on Open Component Ecosystems, ACM. pp. 21–30.

[20] Lehman, M., 1980. Programs, life cycles, and laws of software evolution. Proceedings of the IEEE 68, 1060–1076.

[21] Lehman, M.M., Ramil, J.F., 2000. Software evolution in the age of component-based software engineering. IEEE Proceedings 147, 249–255.

[22] Liaskos, S., Lapouchnian, A., Wang, Y., Yu, Y., Easterbrook, S., 2005. Configuring common personal software: a requirements-driven approach, in: Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on, pp. 9 – 18.

[23] Mancinelli, F., Boender, J., Di Cosmo, R., Vouillon, J., Durak, B., Leroy, X., Treinen, R., 2006. Managing the complexity of large free and open source package-based software distributions, in: ASE 2006: Automated Software Engineering, IEEE. pp. 199–208.

[24] Mancoosi, 2010. Mancoosi software tools. http://www.mancoosi.org/software/. Retrieved December 2010.

[25] Massol, V., O'Brien, T.M., 2005. Maven: A Developer's Notebook. O'Reilly Media.

[26] Michel, C., Rueher, M., 2010. Handling software upgradeability problems with MILP solvers, in: LoCoCo 2010: Logics for Component Configuration, pp. 1–10.

[27] Narayanaswamy, K., Scacchi, W., 1987. Maintaining configurations of evolving software systems. IEEE Transactions on Software Engineering 13, 324–334.

[28] Noronha Silva, G., 2008. APT howto. http://www.debian.org/doc/manuals/apt-howto/.

[29] OSGi Alliance, 2003. OSGi Service Platform, Release 3. IOS Press, Inc.

[30] Perry, D.E., 1987. Software interconnection models, in: 9th International Conference on Software Engineering, IEEE Computer Society Press. pp. 61–69.

[31] Raymond, E.S., 2001. The cathedral and the bazaar. O'Reilly.

[32] Schaefer, T.J., 1978. The complexity of satisfiability problems, in: 10th Annual ACM Symposium on Theory of Computing, ACM. pp. 216–226.

[33] Schmid, K., 2010. Variability modeling for distributed development — a comparison with established practice, in: SPLC 2010: Software Product Lines Conference, Springer. pp. 151–165.

[34] Steuer, R.E., 1986. Multiple Criteria Optimization: Theory, Computation and Application. Wiley.

[35] Stokely, M., 2004. The FreeBSD Handbook. FreeBSD Mall. 3 edition.

[36] Szyperski, C., 1998. Component Software. Beyond Object-Oriented Programming. Addison-Wesley.

[37] Treinen, R., Zacchiroli, S., 2009a. Common Upgradeability Description Format (CUDF) 2.0. Technical Report 3. The Mancoosi Project. http://www.mancoosi.org/reports/tr3.pdf.

[38] Treinen, R., Zacchiroli, S., 2009b. Expressing advanced user preferences in component installation, in: IWOCE 2009: International Workshop on Open Component Ecosystems, ACM. pp. 31–40.

[39] Trezentos, P., Lynce, I., Oliveira, A., 2010. Apt-pbo: Solving the software dependency problem using pseudo-boolean optimization, in: ASE'10: Automated Software Engineering, ACM. pp. 427–436.

[40] Tucker, C., Shuffelton, D., Jhala, R., Lerner, S., 2007. OPIUM: Optimal package install/uninstall manager, in: ICSE'07: International Conference on Software Engineering, IEEE. pp. 178–188.

[41] Vieira, M., Richardson, D., 2002. The role of dependencies in component-based systems evolution, in: IWPSE'02: International Workshop on Principles of Software Evolution, ACM. pp. 62–65.

Appendix A. CUDF syntax

Overall structure.

$$\mathit{cudf} ::= \mathit{preamble}?\ \mathit{universe}\ \mathit{request}$$

Flow elements.

$$\begin{aligned}
\mathit{ssep} &::= (\mathit{comment} | \text{`}\texttt{\textbackslash n}\text{'})* \ \text{`}\texttt{\textbackslash n}\text{'}\ (\mathit{comment} | \text{`}\texttt{\textbackslash n}\text{'})* \\
\mathit{comment} &::= \text{`}\texttt{\#}\text{'}\ \mathit{line} \\
\mathit{line} &::= \texttt{[\^{}\textbackslash n]} * \ \text{`}\texttt{\textbackslash n}\text{'}
\end{aligned}$$

Document parts.

$$\begin{aligned}
\mathit{preamble} &::= \text{`}\texttt{preamble: }\text{'}\ \mathit{line}\ \mathit{stanza}\ \mathit{ssep} \\
\mathit{universe} &::= \mathit{package}* \\
\mathit{package} &::= \text{`}\texttt{package: }\text{'}\ \mathit{pkgname}\ \mathit{stanza}\ \mathit{ssep} \\
\mathit{request} &::= \text{`}\texttt{request: }\text{'}\ \mathit{line}\ \mathit{stanza}\ \mathit{comment}*
\end{aligned}$$

Stanzas.

$$\begin{aligned}
\mathit{stanza} &::= (\mathit{property}\ \text{`}\texttt{\textbackslash n}\text{'} \mid \mathit{comment})* \\
\mathit{property} &::= \mathit{propname}\ \text{`}\texttt{: }\text{'}\ \mathit{value} \\
\mathit{propname} &::= \mathit{ident} \\
\mathit{value} &::= \mathit{bool} \mid \mathit{enum} \mid \mathit{int} \mid \mathit{nat} \mid \mathit{posint} \mid \mathit{string} \mid \mathit{pkgname} \mid \mathit{ident} \mid \mathit{typedecl} \\
&\quad \mid \mathit{vpkg} \mid \mathit{veqpkg} \mid \mathit{vpkgformula} \mid \mathit{vpkglist} \mid \mathit{veqpkglist}
\end{aligned}$$

Values: CUDF types.

$$\begin{aligned}
\mathit{bool} &::= \text{`}\texttt{true}\text{'} \mid \text{`}\texttt{false}\text{'} \\
\mathit{int} &::= (\text{`}\texttt{+}\text{'} | \text{`}\texttt{-}\text{'})?\ \texttt{[0-9]}+ \\
\mathit{string} &::= \texttt{[\^{}\textbackslash r\textbackslash n]}* \\
\mathit{vpkg} &::= \mathit{pkgname}\ (\mathit{sp}+\ \ \mathit{vconstr})? \\
\mathit{vpkgformula} &::= \mathit{andfla} \mid \text{`}\texttt{true!}\text{'} \mid \text{`}\texttt{false!}\text{'} \\
\mathit{vpkglist} &::= \text{`'} \mid \mathit{vpkg}\ (\mathit{sp}*\ \text{`}\texttt{,}\text{'}\ \mathit{sp}*\ \mathit{vpkg})* \\
\mathit{enum} &::= \mathit{ident} \\
\mathit{pkgname} &::= \texttt{[A-Za-z0-9+./@()\%-]}+ \\
\mathit{ident} &::= \texttt{[a-z]}\,\texttt{[a-z0-9-]}* \\
\mathit{nat} &::= \text{`}\texttt{+}\text{'}\texttt{[0-9]}+ \\
\mathit{posint} &::= \text{`}\texttt{+}\text{'}\texttt{[0-9]}*\ \texttt{[1-9]}\,\texttt{[0-9]}* \\
\mathit{veqpkg} &::= \mathit{pkgname}\ (\mathit{sp}+\ \ \mathit{veqconstr})? \\
\mathit{veqpkglist} &::= \text{`'} \mid \mathit{veqpkg}\ (\mathit{sp}*\ \ \text{`}\texttt{,}\text{'}\ \ \mathit{sp}*\ \ \mathit{veqpkg})* \\
\mathit{typedecl} &::= \text{`'} \mid \mathit{typedecl1}\ \ (\mathit{sp}*\ \ \text{`}\texttt{,}\text{'}\ \ \mathit{sp}*\ \ \mathit{typedecl1})*
\end{aligned}$$

Value: gory details.

```
vconstr ::= reop sp+ ver
veqconstr ::= '=' sp+ ver
relop ::= '=' | '!=' | '>=' | '>' | '<=' | '<'
sp ::= ' ' | '\t'
ver ::= posint
andfla ::= orfla (sp* ',' sp* orfla)*
orfla ::= atomfla (sp* '|' sp* atomfla)*
atomfla ::= vpkg
typedecl1 ::= ident sp* ':' sp* typeexpr(sp* = sp* '[' value* ']')?
typeexpr ::= typename | 'enum' sp* '[' ident (',' sp* ident)* ']'
typename ::= 'bool' | 'int' | 'nat' | 'posint' | 'string' | 'pkgname'
           | 'ident' | 'vpkg' | 'veqpkg' | 'vpkgformula' | 'vpkglist'
           | 'veqpkglist'
```

Appendix B. CUDF semantics

Appendix B.1. CUDF types

We start by defining the domains of CUDF types, which are used in the definition of the semantics later on.

Definition 1 (CUDF type domains)**.**

- $\mathcal{V}(\texttt{posint})$ *is the set of* positive *natural numbers*
- $\mathcal{V}(\texttt{ident})$ *is a set of distinguished labels (intuitively, there is one such label for each lexically valid CUDF identifier)*
- $\mathcal{V}(\texttt{bool})$ *is the set* $\{\texttt{true}, \texttt{false}\}$
- $\mathcal{V}(\texttt{vpkgformula})$ *is the smallest set* F *such that:*

$$\begin{array}{lll}
\texttt{true} \in F & & \textit{(truth)} \\
\texttt{false} \in F & & \textit{(untruth)} \\
\mathcal{V}(\texttt{vpkg}) \subseteq F & & \textit{(package predicate)} \\
\bigvee_{i=1,\ldots,n} a_i \in F & a_1, \ldots, a_n \ \textit{atoms} \in F & \textit{(disjunctions)} \\
\bigwedge_{i=i,\ldots,n} d_i \in F & d_1, \ldots, d_n \ \textit{disjunctions} \in F & \textit{(conjunctions)}
\end{array}$$

- $\mathcal{V}(\texttt{vpkglist})$ *is the smallest set* L *such that:*

$$\begin{array}{lll}
[] \in L & & \textit{(empty lists)} \\
p{::}l \in L & p \in \mathcal{V}(\texttt{vpkg}), l \in L & \textit{(package concatenations)}
\end{array}$$

- $\mathcal{V}(\texttt{veqpkglist})$ *is the smallest set* $L' \subseteq \mathcal{V}(\texttt{vpkglist})$ *such that:*

$$\begin{array}{lll}
[] \in L' & & \textit{(empty lists)} \\
p{::}l \in L' & p \in \mathcal{V}(\texttt{veqpkg}), l \in L' & \textit{(package concatenations)}
\end{array}$$

Appendix B.2. CUDF formal semantics

CUDF semantics is defined in a style similar to [23], however, we now have to deal with an abstract semantics that is closer to "real" problem descriptions, and that contains artifacts like *features*. This induces some complications for the definition of the semantics. In [23] this and similar problems were avoided by a pre-processing step that expands many of the notions that we wish to keep in the CUDF format.

Appendix B.3. Abstract syntax and semantic domains

The abstract syntax and the semantics is defined using the value domains defined in Appendix B.1. In addition, we give the following definitions:

Definition 2.

- $\textsc{Constraints}$ *is the set of version constraints, consisting of the value* $\top$ *and all pairs* $(relop, v)$ *where relop is one of* $=, \neq, <, >, \leq, \geq$ *and* $v \in \mathcal{V}(\texttt{posint})$.
- $\textsc{Keepvalues}$ *is the set of the possible values of the* **keep** *property of package information items, that is:* $\{\texttt{version}, \texttt{package}, \texttt{feature}, \texttt{none}\}$

The abstract syntax of a CUDF document is a pair consisting of a package description (as defined in Definition 3) and a request (see Definition 5).

Definition 3 (Package description)**.** *A* package description *is a partial function*

$$\mathcal{V}(\texttt{ident}) \times \mathcal{V}(\texttt{posint}) \rightsquigarrow \mathcal{V}(\texttt{bool}) \times \textsc{Keepvalues} \times \mathcal{V}(\texttt{vpkgformula}) \times \mathcal{V}(\texttt{vpkglist}) \times \mathcal{V}(\texttt{veqpkglist})$$

The set of all package descriptions is noted $\textsc{Descr}$. *If* ϕ *is a package description then we write* $Dom(\phi)$ *for its domain. If* $\phi(p, n) = (i, k, d, c, p)$ *then we also write*

- $\phi(p, n).\texttt{installed} = i$
- $\phi(p, n).\texttt{keep} = k$
- $\phi(p, n).\texttt{depends} = d$
- $\phi(p, n).\texttt{conflicts} = c$
- $\phi(p, n).\texttt{provides} = p$

It is natural to define a package description as a function since we can have at most one package description for a given pair of package name and version in a CUDF document. The function is generally only partial since we clearly do not require to have a package description for any possible pair of package name and version.

We define the removal operation of a particular versioned package from a package description. This operation will be needed later in Definition 14 to define the semantics of *package conflicts* in case a package conflicts with itself or a feature provided by the same package.

Definition 4 (Package removal)**.** *Let ϕ be a package description, $p \in \mathcal{V}(\texttt{ident})$ and $n \in \mathcal{V}(\texttt{posint})$. The package description $\phi - (p,n)$ is defined by*

$$\begin{array}{rcl} Dom(\phi - (p,n)) & = & Dom(\phi) - \{(p,n)\} \\ (\phi - (p,n))(q,m) & = & \phi(q,m) \qquad \textit{for all } (q,m) \in Dom(\phi - (p,n)) \end{array}$$

Definition 5 (Request)**.** *A* request *is a triple (l_i, l_u, l_d) with $l_i, l_u, l_d \in \mathcal{V}(\texttt{vpkglist})$.*

In a triple (l_i, l_u, l_d), l_i is the list of packages to be installed, l_u the list of packages to be updated, and l_d the list of packages to be deleted.

Appendix B.4. Installations

Definition 6 (Installation)**.** *An* installation *is a function from $\mathcal{V}(\texttt{ident})$ to $P(\mathcal{V}(\texttt{posint}))$.*

The idea behind this definition is that the function describing an installation associates the set of versions that are installed to any possible package name. This set is empty when no version of the package is installed.

We can extract an installation from any package description as follows:

Definition 7 (Current installation)**.** *Let ϕ be a package description, the* current package installation of ϕ

$$i_\phi \colon \mathcal{V}(\texttt{ident}) \to P(\mathcal{V}(\texttt{posint}))$$

is defined by

$$i_\phi(p) := \{n \in \mathcal{V}(\texttt{posint}) \mid (p,n) \in Dom(\phi) \textit{ and } \phi(p,n).\texttt{installed} = \texttt{true}\}$$

A package can declare zero or more *features* that it provides. The function f_ϕ defined below associates to any package name (here intended to be a the name of a virtual package) the set of version numbers with which this virtual package is provided by some of the packages installed by ϕ:

Definition 8 (Current features)**.** *Let ϕ be a package description, the* current features of ϕ

$$f_\phi \colon \mathcal{V}(\texttt{ident}) \to P(\mathcal{V}(\texttt{posint}))$$

is defined by

$$\begin{array}{rll} f_\phi(p) & := \{n \in \mathcal{V}(\texttt{posint}) \mid & \textit{exists } q \in Dom(i_\phi) \textit{ exists } m \in i_\phi(q) \textit{ such that} \\ & & (((=,n),p) \in \phi(q,m).\texttt{provides} \textit{ or } (\top,p) \in \phi(q,m).\texttt{provides})\} \end{array}$$

The second case in the definition above expresses the fact that providing a feature without a version number means providing that feature at any possible version.

In order to define the semantics of a CUDF document, we will frequently need to merge two installations. This will mainly be used for merging an installation of packages with an installation of provided features. The merging operation is formalized as follows:

Definition 9 (Merging)**.** *Let $f, g \colon \mathcal{V}(\texttt{ident}) \to P(\mathcal{V}(\texttt{posint}))$ be two installations. Their merge $f \cup g \colon \mathcal{V}(\texttt{ident}) \to P(\mathcal{V}(\texttt{posint}))$ is defined as*

$$(f \cup g)(p) = f(p) \cup g(p) \qquad \textit{for any } p \in \mathcal{V}(\texttt{ident})$$

Appendix B.5. Consistent package descriptions

We define what it means for an installation to satisfy a constraint:

Definition 10 (Constraint satisfaction)**.** *The* satisfaction *relation between a natural number n and a constraint $c \in$ Constraints, noted $n \models c$, is defined as follows:*

$n \models \top$	*for any n*	$n \models (<, v)$	*iff $n < v$*
$n \models (=, v)$	*iff $n = v$*	$n \models (>, v)$	*iff $n > v$*
$n \models (\neq, v)$	*iff $n \neq v$*	$n \models (\leq, v)$	*iff $n \leq v$*
		$n \models (\geq, v)$	*iff $n \geq v$*

Now we can define what it implies for a package installation to satisfy some formula:

Definition 11 (Formula satisfaction)**.** *The* satisfaction *relation between an installation I and a formula p, noted $I \models p$, is defined by induction on the structure of p:*

- *$I \models (c, p)$ where, $c \in$ Constraints and $p \in \mathcal{V}(\texttt{ident})$, iff there exists an $n \in I(p)$ such that $n \models c$.*
- *$I \models \phi_1 \wedge \ldots \wedge \phi_n$ iff $I \models \phi_i$ for all $1 \leq i \leq n$.*
- *$I \models \phi_1 \vee \ldots \vee \phi_n$ iff there is an i with $1 \leq i \leq n$ and $I \models \phi_i$.*

We can now lift the satisfaction relation to sets of packages:

Definition 12. *Let I be an installation, and $l \in \mathcal{V}(\texttt{vpkglist})$. Then $I \models l$ if for any $(c, p) \in l$ there exists $n \in I(p)$ with $n \models c$.*

Note that, given that $\mathcal{V}(\texttt{veqpkglist}) \subseteq \mathcal{V}(\texttt{vpkglist})$, this also defines the satisfaction relation for elements of $\mathcal{V}(\texttt{veqpkglist})$. Also note that one could transform any $l \in \mathcal{V}(\texttt{vpkglist})$ into a formula $l_\wedge \in \mathcal{V}(\texttt{vpkgformula})$, by constructing the conjunction of all the elements of l. The semantics of l is the same as the semantics of the formula $l_\wedge$.

Definition 13 (Disjointness)**.** *The* disjointness *relation between an installation I and a set $l \in \mathcal{V}(\texttt{vpkglist})$ of packages possibly with version constraints, is defined as: $I \parallel l$ if for any $(c, p) \in l$ and all $n \in I(p)$ we have that $n \not\models c$.*

Definition 14. *A package description ϕ is* consistent *if for every package $p \in \mathcal{V}(\texttt{ident})$ and $n \in i_\phi(p)$ we have that*

1. $i_\phi \cup f_\phi \models \phi(p, n).\texttt{depends}$
2. $i_{\phi-(p,n)} \cup f_{\phi-(p,n)} \parallel \phi(p, n).\texttt{conflicts}$

In the above definition, the first clause corresponds to the *Abundance* property of [23]: all the dependency relations of all installed packages must be satisfied. The second clause corresponds to the *Peace* property of [23]. In addition, we now have to take special care of packages that conflict with themselves, or that provide a feature and at the same time conflict with that feature: we only require that there be no conflict with any *other* installed package and with any feature provided by some *other* package (see also Section Appendix B.7).

Appendix B.6. Semantics of requests

The semantics of a request is defined as a relation between package descriptions. The idea is that two package descriptions ϕ_1 and ϕ_2 are in the relation defined by the request r if there exists a transformation from ϕ_1 to ϕ_2 that satisfies r.[9]

First we define the notion of a successor of a package description:

Definition 15 (Successor relation)**.** *A package description ϕ_2 is called a* successor *of a package description ϕ_1, noted $\phi_1 \rightarrowtail \phi_2$, if*

1. $Dom(\phi_1) = Dom(\phi_2)$
2. *For all $p \in \mathcal{V}(\texttt{ident})$ and $n \in \mathcal{V}(\texttt{posint})$: if $\phi_1(p,n) = (i_1, k_1, d_1, c_1, p_1)$ and $\phi_2(p,n) = (i_2, k_2, d_2, c_2, p_2)$ then $k_1 = k_2$, $d_1 = d_2$, $c_1 = c_2$, and $p_1 = p_2$.*
3. *For all $p \in \mathcal{V}(\texttt{ident})$*
   - *for all $n \in i_{\phi_1}(p)$: if $\phi_1(p,n).\texttt{keep} = \texttt{version}$ then $n \in i_{\phi_2}(p)$.*
   - *if there is an $n \in i_{\phi_1}(p)$ with $\phi_1(p,n).\texttt{keep} = \texttt{package}$ then $i_{\phi_2}(p) \neq \emptyset$*
   - *for all $n \in i_{\phi_1}(p)$: if $\phi_1(p,n).\texttt{keep} = \texttt{feature}$ then $i_{\phi_2} \cup f_{\phi_2} \models \phi_1(p,n).\texttt{provides}$*

The first and the second item of the above definitions indicate that a successor of a package description ϕ may differ from ϕ only in the status of packages. The third item refines this even further depending on keep values:

- If we have a keep status of `version` for an installed package p and version n then we have to keep that package and version.
- If we have a keep status of `package` for some installed version of a package p then the successor must have at least one version of that package installed.
- If we have a keep status of `feature` for some installed version n of a package p then the successor must provide all the features that where provided by version n of package p.

Definition 16 (Request semantics)**.** *Let $r = (l_i, l_u, l_d)$ be a request. The* semantics *of r is a relation $\overset{r}{\curvearrowright} \subseteq \textsc{Descr} \times \textsc{Descr}$ defined by $\phi_1 \overset{r}{\curvearrowright} \phi_2$ if*

1. $\phi_1 \rightarrowtail \phi_2$
2. *ϕ_2 is consistent*
3. $i_{\phi_2} \cup f_{\phi_2} \models l_i$
4. $i_{\phi_2} \cup f_{\phi_2} \parallel l_d$
5. *$i_{\phi_2} \cup f_{\phi_2} \models l_u$, and for all p such that $(c,p) \in l_u$ we have that $(i_{\phi_2} \cup f_{\phi_2})(p) = \{n\}$ (i.e., is a singleton set) where $n \geq n'$ for all $n' \in (i_{\phi_1} \cup f_{\phi_1})(p)$.*

[9] The definition of optimization criteria will is outside the scope of this document; see Section 5 of the "Dependency Solving: a Separate Concern in Software Evolution Management" manuscript.

Appendix B.7. Comments on the semantics

Installing multiple versions of the same package. The semantics allows a priori to install multiple versions of the same package. This coincides with the semantics found in RPM-like FOSS distributions (which a priori do not forbid to install multiple versions of the same package), but is in opposition to the semantics found in Debian-like FOSS distributions (which allow for one version of any package to be installed at most).

In many practical cases the distinction between a priori allowing or not for multiple versions of a package makes little difference. In the RPM world multiple versions of the same package are very often in a conflict by their features or shipped files. If both versions of the same package provide the same feature and also conflict with that feature then the RPM semantics, as the CUDF semantics, does not allow to install both at the same time. Only packages that have been designed to have distinct versions provide distinct features (in particular, files with distinct paths) can in practice be installed in the RPM world in several different versions at a time. This typically applies to operating system packages. In order to have a meta-installer with Debian semantics work correctly on such a package description, it is sufficient to rename the packages, and to create a new package, say $p - n$, for a package p and version n when p can be installed in several versions.

On the other hand, a meta-installer with RPM semantics will produce solutions on a package description that would not be found by a meta-installer with Debian semantics since it is free to install several version of the same package. The uniqueness restriction of Debian can easily be made explicit in the package description by adding a to each package description stanza, say for package name "p", a serialized property "`conflicts` p".

Upgrading packages. Even though the semantics allows for multiple installed versions of the same package, the notion of "upgrade" (at least for what concerns this specification) is intimately tied to a single installed version of a given package.

Hence, for an upgrade request to be fulfilled for a package p, exactly one version of p must be installed in the resulting package status. Additionally, to preserve the "upgrade" intuition, the resulting installed version must be greater or equal than the *greatest* version of p which was previously installed. Both these conditions are expressed by point (5) of Definition 16. Note that a strictly greater version of what was previously installed can be requested by specifying a suitable "`>`" predicate as part of the `upgrade` property.

Upgrading virtual packages. Virtual packages, or features, can be with or without version specification. The fact that the lack of version specifications is interpreted as providing all possible versions of a given feature (see Definition 8) interacts with the semantic of upgrades when virtual packages are mentioned within `upgrade`. In particular, upgrades are de facto possible only for versioned virtual packages.[10]

[10]The reason is that upgraded (virtual) packages must correspond to singleton sets in the resulting package status, whereas non-versioned virtual packages will provide infinite sets. Similarly, if in the initial package status a virtual package is non-versioned, it will provide an infinite version sets, whose maximum cannot be matched by any singleton set in the resulting package status.

CHAPTER 9

Strong Dependencies between Software Components

This chapter contains the full text of the article "Strong Dependencies between Software Components" *[1].*

Strong Dependencies between Software Components*

Pietro Abate
abate@pps.jussieu.fr
Universitè Paris Diderot, PPS
UMR 7126, Paris, France

Jaap Boender
Jaap.Boender@pps.jussieu.fr

Roberto Di Cosmo
roberto@dicosmo.org

Stefano Zacchiroli
zack@pps.jussieu.fr

Abstract

Component-based systems often describe context requirements in terms of explicit inter-component dependencies. Studying large instances of such systems—such as free and open source software (FOSS) distributions—in terms of declared dependencies between packages is appealing. It is however also misleading when the language to express dependencies is as expressive as boolean formulae, which is often the case. In such settings, a more appropriate notion of component dependency exists: strong dependency. *This paper introduces such notion as a first step towards modeling semantic, rather then syntactic, inter-component relationships.*

Furthermore, a notion of component sensitivity *is derived from strong dependencies, with applications to quality assurance and to the evaluation of upgrade risks. An empirical study of strong dependencies and sensitivity is presented, in the context of one of the largest, freely available, component-based system.*

1. Introduction

Component-based software architectures [21] have the property of being upgradeable piece-wise, without necessarily touching all the pieces at the same time. The more pieces are affected by a single upgrade, the higher the impact of the upgrade can be on the usual operations performed by the overall system; this impact can either be beneficial (if the upgrade works as planned) or disastrous (if not). Package-based FOSS (Free and Open Source Software) distributions are possibly the largest-scale examples of component-based architectures, their upgrade effects are experienced daily by million of users world-wide, and the historical data concerning their evolution is publicly available.

Within FOSS distributions, software components are managed as *packages* [6]. Packages are described with meta-information, which include complex inter-relationships describing the static requirements to run properly on a target system. Requirements are expressed in terms of other packages, possibly with restrictions on the desired versions. Both positive requirements (*dependencies*) and negative requirements (*conflicts*) are usually allowed.

Example 1.1. *An excerpt of the inter-package relationships of the* `postfix` *Internet mail transport agent in Debian GNU/Linux*[1] *currently reads:*

```
1  Package: postfix
2  Version: 2.5.5-1.1
3  Depends: libc6 (>= 2.7), libdb4.6, ssl-cert,
4    libsasl2-2, libssl0.9.8 (>= 0.9.8f-5),
5    debconf (>= 0.5) | debconf-2.0,
6    netbase, adduser (>= 3.48), dpkg (>= 1.8),
7    lsb-base (>= 3.0-6)
8  Conflicts: libnss-db (<< 2.2-3), smail,
9    mail-transport-agent, postfix-tls
10 Provides: mail-transport-agent, postfix-tls
```

As this short example shows, inter-package relationships can get quite complex, and there are plenty of more complex examples to be found in distributions like Debian. In particular, the language to express package relationships is not as simple as *flat* lists of component predicates, but rather a structured language whose syntax and semantics is expressed by conjunctive normal form (CNF) formulae [17]. In Example 1.1, commas represent logical conjunctions among predicates, whereas bars ("|") represent logical disjunctions. Also, indirections by the mean of so-called *virtual packages* can be used to declare feature names over which other packages can declare relationships; in the example (see line 10: "Provides") the package declares to provide the features called `postfix-tls` and `mail-transport-agent`.

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

Within this setting, it is interesting to analyse the *dependency graph* of all packages shipped by a mainstream FOSS distribution. This graph is potentially very large as distributions like Debian are composed of several tens of thousands packages, but it is surely smaller than widely studied graphs such as the World Wide Web graph [1]. It is also more expressive though, in the sense that it contains different types of edges (dependencies and conflicts for example) and allows the use of disjunctions to express alternative paths. Simple encodings of the package universe have been proposed in the past [14, 16], to study the adherence of the dependency graph to small-world network laws. In such encodings, inter-package relationships were approximated by a simple binary relation of *direct* dependency, which is noted $p \to q$ in this paper. Formally, $p \to q$ holds whenever package q occurs syntactically in the dependency formula of p. This notion of direct dependency does not distinguish between q occurring in conjunctive or disjunctive position, ignoring the semantic difference between conjunctive and disjunctive dependencies, as well as the presence of conflicts among components.

In this paper we argue that there is a different dependency graph to be studied to grasp meaningful relationships among software components: a graph that represents the *semantics* of inter-component relationships, in which an edge between two components is drawn only if the first cannot be installed without installing the second. We call such a graph the *strong dependency graph*, argue that it is better suited to study package universes in component-based architectures, and study its network properties. Finally, we argue that the strong dependency graph can be used to establish a measure of package "sensitivity" which has several uses, from distribution wide quality assurance to establishing the potential risks of package upgrades. As a relevant, yet empirical, case study we build and analyse the strong dependency graph of present and past FOSS distributions, as well as the corresponding package sensitivity.

The rest of the paper is structured as follows: Section 2 introduces the notion of strong dependency, highlights the differences with plain dependencies and proposes related sensitivity metrics. Section 3 computes dependencies and sensitivity of components of a large and popular FOSS distribution. Section 4 gives an efficient algorithm to compute strong dependencies for large software repositories. Section 5 discusses applications of the proposed metrics for quality assurance and upgrade risk evaluation. Before concluding, Section 6 discusses related research.

2. Strong dependencies

Component dependencies can be used to compute relevant quality measures of software repositories, for instance to identify particularly fragile components [7, 13, 15]. It is well known that small-world networks are resilient to random failures but particularly weak in the presence of attacks, due to the existence of highly connected *hub nodes* [2]. To identify the components whose modification (e.g., removal or upgrade) can have a high potential impact on the stability of a complex software system, it is natural to look for *hubs* on which a lot of other components depend.

In FOSS distributions, not unlike other component-based systems [3, 4], the language used to encode inter-package relationships is expressive enough to cover propositional logic. As a consequence, considering only *plain connectivity*—i.e., the possibility of going from one package to another following dependency arcs—is no longer meaningful to identify hubs. For example, if p is to be installed and there exists a dependency path from p to q, it is not true that q is always needed for p, and in some cases q may even be incompatible with p.

In other terms, the *syntactic* connectivity notion does not tell much about the real structure of dependencies: we need to go further and analyse the *semantic* connectivity among software components induced by the explicit dependencies in the graph. That has led us to the following definition.

Definition 2.1 (Strong dependency)**.** *Given a repository R, we say that a package p in R* strongly depends *on a package q in R, written $p \Rightarrow_R q$, if there exists a healthy installation of R containing p, and every healthy installation of R containing p also contains q. We write $Spreds(p)_R$ for the set $\{q|q \Rightarrow_R p\}$ of strong predecessors of a package p in R, and $Scons(p)_R$ for the set $\{q \mid p \Rightarrow_R q\}$ of strong successors of p in R.*

In the following, we will drop the R subscript when the repository is clear from the context.

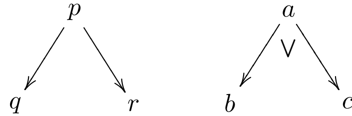
The above notions of repository and healthy installation come from [17]; the underlying intuitions are as follows. A *repository* is a set of packages, together with dependencies and conflicts encoded as propositional logic predicates over other packages contained therein; an *installation* is a subset of the repository; an installation is said to be *healthy* when all its packages have their dependencies satisfied within the installation and dually their conflicts *un*satisfied.

Intuitively, p strongly depends on q with respect to R if it is not possible to install p without also installing q. Notice that the definition requires p to be installable in R as otherwise it would vacuously depend on all the packages q in the repository. Due to the complex nature of dependencies, there can be a huge gap with the syntactic dependency graph as naively extracted from the metadata.

Example 2.2 (Direct vs strong dependencies)**.** *In simple cases, conjunctive direct dependencies translate to identical strong dependencies whereas disjunctive ones vanish, as for the packages of the following repository:*

```
Package: p
Depends: q, r

Package: a
Depends: b | c
```

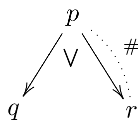


We have that $p \rightarrow q, p \rightarrow r$ and $p \Rightarrow q, p \Rightarrow r$ (because p cannot be installed without either q or r), and that $a \rightarrow b, a \rightarrow c$ whereas $a \not\Rightarrow b, a \not\Rightarrow c$ (because a does not forcibly require neither b nor c). In general however, the situation is much more complex, like in the following repository:

```
Package: p
Depends: q | r

Package: r
Conflicts: p

Package: q
```



Notice that $p \Rightarrow q$ in spite of q not being a conjunctive dependency of p, and r is incompatible with p, despite the fact that $p \rightarrow r$.

Proposition 2.3 (Transitivity)**.** *If $p \Rightarrow_R q$ and $q \Rightarrow_R r$ then $p \Rightarrow_R r$.*

Proof. Trivial from Definition 2.1. □

On top of the strong and direct dependency notions, we can define the corresponding *dependency graphs*.

Definition 2.4 (Dependency graphs)**.** *The* strong dependency graph *$SG(R)$ of a repository R is the directed graph having as vertices the packages in R and as edges all pairs $\langle p, q\rangle$ such that $p \Rightarrow q$. Note that the $SG(R)$ is transitively closed as direct consequence as the transitivity of the strong dependency relation.*

Similarly, the direct dependency graph *$DG(R)$ is the directed graph having as vertices the packages in R and as edges all pairs $\langle p, q\rangle$ such that $p \rightarrow q$.*

The dependency graphs can be used to formalise, via the notion of *impact set*, the intuitive notion of the set of packages which are potentially affected by changes in a given package.

Definition 2.5 (Impact set of a component)**.** *Given a repository R and a package p in R, the* impact set *of p in R is the set $Is(p, R) = \{q \in R \mid q \Rightarrow p\}$.*

Similarly, the direct impact set *of p is the set $DirIs(p, R) = \{q \in R \mid q \rightarrow p\}$.*

While the impact set gives a sound lower bound to the set of packages which can be potentially affected by a change in a package, the direct impact set offers no similar guarantees. Note that by Definition 2.1, for all package p, $p \in Is(p, R)$. Package sensitivity—a measure of how sensitive is a package, in terms of how many other packages can be affected by a change in it—can now be defined as follows.

Definition 2.6 (Sensitivity)**.** *The strong sensitivity, or simply* sensitivity, *of a package $p \in R$ is $|Is(p, R)| - 1$, i.e., the cardinality of the impact set minus 1.*[2]

Similarly, the direct sensitivity *is the cardinality of the direct impact set.*

The higher the sensitivity of a package p, the higher the *minimum* number of packages which will be potentially affected by a change, such as a new bug, introduced in p. We write $|p|$ and $||p||$ to denote the direct and strong sensitivity of package p, respectively. The following basic property of impact sets and sensitivity follows easily from the definitions.

Proposition 2.7 (Inclusion of impact sets)**.** *If $p \Rightarrow_R q$ then $Is(p, R) \subseteq Is(q, R)$. As a consequence, the sensitivity of p in R is smaller than the sensitivity of q in R.*

When analysing a large component base, like Debian's, which contains about 22,000 components, it is important to be able to identify some measure that can be used to easily pinpoint "interesting" packages. Sensitivity can be (and actually is, in our tools) used to order packages, bringing the most sensitive to the forefront. To this end is important to note that (strong) sensitivity can be computed *automatically* (and efficiently, see Section 4) from dependencies; that is an important feature: given the sheer size of systems like Debian, it would be unreasonable to try mix sensitivity with hand-maintained classifications such as "core" packages, "end-user" packages, etc. But sensitivity alone is not enough: we do not want to spend time going through hundreds of packages with similar sensitivity to find the one which is really important, so we need to keep some of the structure of the strong dependency graph.

A first step is to group together only those packages that are related by strong dependencies, but our analysis of the Debian distribution led us to discover that we really need to go further and distinguish the cases of related components in the strong dependency graph from the cases of unrelated ones: in the picture in Figure 1,[3] configuration 1c shows q that clearly dominates r, as the impact set of r really comes from that of q, in configuration 1d, q and r are clearly equivalent, while in configuration 1a, q and r are totally unrelated, and in configuration 1b, q strong depends on r but q does not generate all the impact set of r.

[2] The -1 accounts for the fact that the impact set of a package always contains itself. This way we ensure that sensitivity 0 preserves the intuitive meaning of "no package potentially affected".

[3] Edges implied by transitivity are omitted from the diagrams for the sake of clarity.

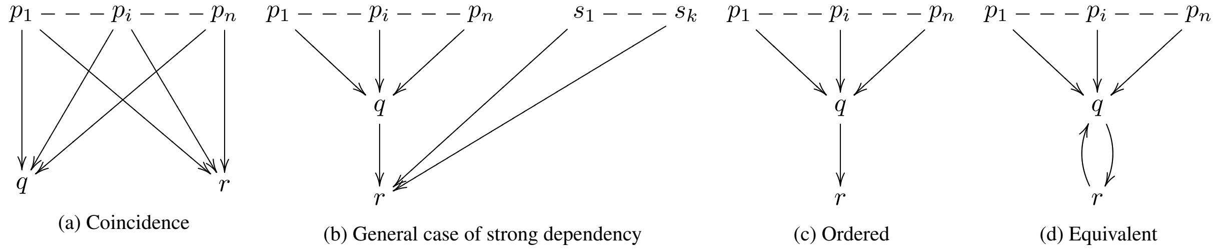


Figure 1. Significant configurations in the strong dependency graph

Yet, the packages q and r all have essentially the same sensitivity values (n or $n+1$) in all the first three cases (and $n+k$ in the fourth, which can also contribute to the mass of packages of sensitivity similar to n). To distinguish these different configurations in strong dependency graphs, we introduce one last notion.

Definition 2.8 (Strong dominance)**.** *Given two packages p and q in a repository R, we say that p strongly dominates q ($p \succeq_{Is} q$) iff*

- *$Is(p,R) \supseteq (Is(q,R) \setminus Scons(p))$, and*
- *p strongly depends on q*

The intuition of strong dominance, is that a package p dominates q if the strong dependency of p on q "explains" the impact set of q: the packages that q has an impact on are really those that p has an impact on, plus p. This notion has some similarity in spirit with the standard notion of dominance used in control flow graphs, but is technically quite different, as strong dependency graphs are transitive, and have no single start node.

Using the transitivity of strong dependencies, the following can be established.

Proposition 2.9. *The strong domination relation is a partial pre-order.*

Proof. Reflexivity is trivial to check. For transitivity, suppose we have $p \succeq_{Is} q$ and $q \succeq_{Is} r$: first of all, p strongly depends on r is a direct consequence of the fact that the strong dependency relation is transitive, so the second condition for $p \succeq_{Is} r$ is established. For the first condition, we know that $Is(p,R) \supseteq (Is(q,R) \setminus Scons(p))$ and $Is(q,R) \supseteq (Is(r,R) \setminus Scons(q))$. By transitivity of strong dependencies, since $p \Rightarrow q \Rightarrow r$, we also have that $Scons(p) \supseteq Scons(q) \supseteq Scons(r)$. Then we have easily that $Is(p,R) \supseteq (Is(q,R) \setminus Scons(p)) \supseteq (Is(r,R) \setminus Scons(q)) \setminus Scons(p) = Is(r,R) \setminus Scons(p)$. □

This pre-order is now able to distinguish among the cases of Figure 1. In Figure 1c we have that $q \succeq_{Is} r$, but not the converse; in 1d both $q \succeq_{Is} r$ and $r \succeq_{Is} q$ hold, i.e., q and r are equivalent according to strong domination; in 1a and 1b no dominance relationship can be established between q and r.

It is possible, and actually quite useful, to generalise the strong dominance relation to cover also the case shown in 1b, where a part of the impact set of the package r is not covered by the impact set of q, as follows.

Definition 2.10 (Relative strong dominance)**.** *Given two packages p and q in a repository R, we say that p strongly dominates q up to z ($p \succeq^z_{Is} q$) iff*

- *$\frac{|(Is(q,R) \setminus Scons(p)) \setminus Is(p,R)|}{|Is(p,R)|} * 100 = z$, and*
- *p strongly depends on q*

It is easy to see that $p \succeq_{Is} q$ iff $p \succeq^0_{Is} q$, and one can compute in a single pass on the repository the values z for each pair of packages such that $p \Rightarrow q$, leaving for later the choice of a threshold value for z. In the case of figure 1b, we have that q dominates r up to $k/n * 100$.

3. Strong dependencies in Debian

Due to the different properties of direct and strong dependencies, the two measures of package sensitivity can differ substantially. To verify that, as well as other properties of the underlying dependency graphs, we have chosen Debian GNU/Linux as a case study.[4] The choice is not casual: Debian is the largest FOSS distribution in terms of number of packages (about $22,000$ in the latest stable release) and, to the best of our knowledge, the largest component-based system freely available for study.

All stable releases of Debian have been considered, from 1994 to February 2009. For each release the archive section `main` and in particular the `i386` architecture has been

[4] The data presents in this section, as well as what was omitted due to space constraints, are available to download from `http://www.mancoosi.org/data/strongdeps/`. The tools used to compute the data are released under open source licenses and are available from the Subversion repository at `https://gforge.info.ucl.ac.be/svn/mancoosi/`.

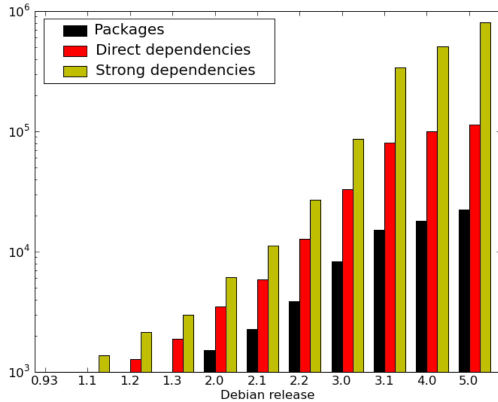


Figure 2. Evolution of packages, direct, and strong dependencies in Debian releases.

considered; the choices are justified by the fact that they identify both the most used parts of Debian,[5] and that they are the only parts which have been part of all Debian releases and hence can be better compared over time. The obtained archive parts have been analysed by building both the direct and strong dependency graphs; while the construction of the former is a trivial exercise, the implemented efficient way of constructing the latter is discussed in Section 4. To build the direct dependency graph the `Depends` and `Pre-Depends` inter-package relationships have been considered [12].

Figure 2 shows the resulting evolution of the number of graph nodes and edges across all Debian releases. The size of the distribution has grown steadily, yet super-linearly, across most releases [20, 11], but the growth rate has decreased in the past two releases. As expected, strong and direct sensitivity are not entirely unrelated, given that the former is the semantic view of the latter, hence they tend to grow together.

More precisely the total number of strong dependencies is higher, in all releases, than the total number of direct dependencies. A partial explanation comes from the fact that the strong dependency graph is a transitive closed graph—property inherited by the underlying strong dependency relationship—whereas the direct dependency graph is not. Performing the transitive closure of the direct dependency graph however would be meaningless, because the propagation rules of disjunctive and conjunctive dependencies are not expressible simply in terms of transitive arcs.

We have studied the apparent correlation between strong and direct dependencies analysing the respective sensitivity measures for each release. Table 1 confirms the correlation and gives some statistical data about package sensitivity. The first column is the Spearman ρ correlation index,[6] a commonly used non-parametric correlation index that is not sensible to exceptional values [8]. An index between 0.5 and 1.0—in all the releases we have $\rho \in [0.91, 0.94]$—is commonly interpreted as a strong correlation between the two variables. The more common correlation index r for the same set of data (not shown in the table) gives consistently a value of 0.55: the huge difference among ρ and r indicates that the few exceptional values in the data series have really high weight; when analyzing some of these exceptional values, we will see how this is indeed the case.

The remaining columns show mean and standard deviation for, respectively, direct sensitivity, strong sensitivity, and $\Delta = ||p|| - |p|$. In particular we note an increasingly high standard deviation in latest Debian releases, which hints that there is an increasing number of peaks.

Figure 3 shows in more detail the correlation phenomenon for Debian 5.0 “Lenny”, the latest (and largest) Debian release. The figure plots strong vs direct sensitivity for each package in the release. In most cases, strong sensitivity is higher than direct sensitivity, yet close: 82.9% of the packages fall in a standard deviation interval from the mean of Δ; the next percentile ranks are 97.4% for two standard deviations, and 99.8% for three. The remaining cases allow for important exceptions of packages with very high strong sensitivity and very low direct sensitivity. Such exceptions are extremely relevant: metrics built on direct sensitivity only would totally overlook packages with a huge potential impact.

Table 1. Direct and strong sensitivity in Debian: correlation, mean, standard deviation.

Rel.	ρ	$\lvert\cdot\rvert$	$\lvert\lvert\cdot\rvert\rvert$	Δ
`.93`	.92	1.00, σ2.79	1.05, σ4.73	1.00, σ4.00
`1.1`	.93	1.70, σ13.9	2.90, σ25.9	1.88, σ18.5
`1.2`	.91	1.79, σ18.4	2.99, σ32.2	1.73, σ22.4
`1.3`	.91	1.92, σ21.9	3.06, σ38.2	1.69, σ25.8
`2.0`	.93	2.29, σ26.7	4.03, σ50.8	2.50, σ36.5
`2.1`	.94	2.60, σ34.9	4.93, σ64.5	2.93, σ46.6
`2.2`	.92	3.29, σ44.2	6.89, σ90.4	4.88, σ68.7
`3.0`	.92	3.99, σ59.2	10.4, σ131.	8.02, σ92.3
`3.1`	.92	5.29, σ91.4	22.3, σ282.	19.3 , σ246.
`4.0`	.92	5.55, σ85.1	28.2, σ352.	24.5 , σ313.
`5.0`	.93	5.07, σ86.1	36.0, σ480.	32.5 , σ440.

[5] According to the Debian popularity contest, available at `http://popcon.debian.org`

[6] The statistical info for the first two rows are possibly not relevant, due to the small size of the two releases.

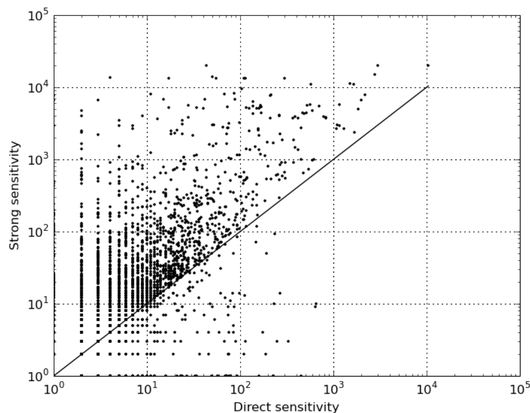


Figure 3. Correlation between strong and direct sensitivity in Debian 5.0

Table 2. Packages from Debian 5.0, sorted by gap between strong / direct impact set sizes.

#	Package	$\|p\|$	$\|\|p\|\|$	$\|\|p\|\| - \|p\|$
1	`gcc-4.3-base`	43	20128	20085
2	`libgcc1`	3011	20126	17115
3	`libselinux1`	50	14121	14071
4	`lzma`	4	13534	13530
5	`coreutils`	17	13454	13437
6	`dpkg`	55	13450	13395
7	`libattr1`	110	13489	13379
8	`libacl1`	113	13467	13354
9	`perl-base`	299	13310	13011
10	`libstdc++6`	2786	14964	12178
11	`libncurses5`	572	11017	10445
12	`debconf`	1512	11387	9875
13	`libc6`	10442	20126	9684
14	`libdb4.6`	103	9640	9537
15	`zlib1g`	1640	10945	9305
16	`debianutils`	86	8204	8118
17	`libgdbm3`	68	8148	8080
18	`sed`	11	8008	7997
19	`ncurses-bin`	1	7721	7720
20	`perl-modules`	214	7898	7684
21	`lsb-base`	211	7720	7509
22	`libxdmcp6`	15	6782	6767
23	`libxau6`	42	6795	6753
24	`libx11-data`	1	6693	6692
25	`libxcb-xlib0`	3	6695	6692
26	`libxcb1`	87	6778	6691
27	`x11-common`	137	6317	6180
28	`perl`	2169	7898	5729
29	`libmagic1`	28	5585	5557
30	`libpcre3`	164	5668	5504

...

3.1. Strong vs direct sensitivity: exceptions

It's time now to look at some of these exceptional cases to see how relevant they are. Table 2 lists the top 30 packages of Lenny having the largest Δ.

`libc6` is the package shipping the C standard library which is required, directly or not, by almost all applications written or otherwise linked to the C programming language. About a half of all the packages in the distribution depends *directly* on `libc6`, as can be seen in row 13 of the table, but almost all packages in the archive cannot be installed without it, as the strong sensitivity of `libc6` is 20'126, on a total of 22'311 packages. In this case direct sensitivity does not inhibit identifying the package as a sensitive one, though, even if it underestimates widely its importance.

Now consider row 1 of Table 2: `gcc-4.3-base`, which is a package without which `libc6` cannot be installed. It is the package with the largest Δ, having direct sensitivity of only 43 and strong sensitivity of 20'128. Ranking its sensitivity with the direct metric would have led to completely miss its importance: a bug into it can potentially affect all packages in the distribution. Note however that `gcc-4.3-base` is not a direct dependency of `libc6`, showing once more that to grasp this kind of interpackage relationships the semantics, rather than the syntax, of dependencies must be put into play.

In the second row, `libgcc1` shows a similar pattern, being this time a direct dependency of `libc6`. The third row and many others in the table show more complex patterns. Ordering packages only according to sensitivity might lead to oversee other important characteristic. Possibly the most extreme cases are those of `ncurses-bin` and `libx11-data`, which are mentioned just once in all the explicit dependencies, and yet are really necessary for several thousand other packages.

We believe this is sufficiently conclusive evidence to totally dismiss, from now on, any analysis based on the syntactic direct dependency graph, when considering component based systems with expressive dependency languages.

3.2. Using strong dominance to cluster data

Now we turn to the problem of presenting the sensitiveness information in a relevant way to a Quality Assurance team: we could simply print a list of package names, ordered by their sensitiveness; this would give a result quite similar to that of table 2 above, just dropping the first and fourth column. A smart Debian developer will surely spot the fact that `gcc-4.3-base`, `libgcc1` and `libc6` are

Table 3. Small-world figures for Debian 5.0.

	Direct dep. graph	Strong dep. graph
Vertices	22,311	22,311
Edges	107,796	40,074
Average degree	4.83	1.80
Clustering coeff.	0.41	0.39
Average distance	3.18	2.86
Components (WCCs)	1,425	2,809
Largest WCC	20,831	19,200
Density	0.00022	0.000081

related and would look at them together, but it would be difficult to see relationships among the other packages in the list, even if we can see that many packages have impact sets of similar size.

Here is where our definition of relative strong dominance comes into play, allowing to build meaningful clusters that provide sensible information to the maintainers: Figure 4 shows the graph of relative strong domination between the first 20 packages of Table 2. Bold edges show strong domination as defined in Definition 2.8. Normal edges show relative domination, where the install sets of the two packages almost fully overlap, apart from a few packages (edges are labelled with the percentage z of Definition 2.10).

This figure shows clearly that it is possible to isolate five clusters of related packages with similar sensitivity values; some of them may look surprising at first sight to a Debian developer, and evident after a little time spent exploring the package metadata: this actually confirms the real value of this way of presenting data.

3.3. Debian is a small world

We expected the strong dependency graph to retain the small world characteristics previously established for the direct dependency graph [14], but this required some extra effort to get sensible results: indeed, computing clustering coefficients and other similar measures on the strong dependency graph will yield very different values (as the strong dependency graph is transitive), so we first built a non-transitive version of the strong dependency graph, and computed the usual small world measures on it.

Note that, since the strong dependency graph contains some cycles, the obtained non-transitive graph is not unique. The differences are however minor enough to not alter the overall results.

The clustering coefficient and average path length of the non-transitive graph are, though slightly smaller, well within the range of small-world networks. More than half the edges of the direct graph have disappeared, but this has not significantly affected either the graph clustering or the path length. The relevant statistics are summarised in Table 3.3.

Some additional notes about obtained small-world statistics. First, both graphs contain one enormous (weakly connected) component, next to which all other components are of insignificant size (for the direct graph, there are 1'480 remaining packages in 1'424 components, which would make their average size just above 1; the ratio is similar for the strong graph). Second, when we look at the density of both graphs (the number of edges in the graph divided by the maximum possible number of edges), we see that both graphs are extremely sparse.

4. Efficient computation

It is not evident that strong dependencies as defined in Section 2 are actually tractable in practise: from previous results [17, 5] it is known that checking installability of a package (or co-installability of a set of packages) is an NP-complete problem. Even if in practise checking installability turns out to be tractable on real-world problem instances, the sheer number of instances that computing strong dependencies may require in the general case makes the problem much harder. We start by observing that the problem of determining strong dependencies is decidable.

Proposition 4.1 (Decidability)**.** *Strong dependencies for packages in a* finite *repository R are* computable.

Proof. Since R is finite, the set of all installations is also finite. Among these installations, finding the healthy one is just a matter of verifying locally the dependency relations. Then, for each p and q, it is enough to check all healthy installations to see whether q is present whenever p is. □

If we want to know if a particular packages p strongly depends on q in a repository R however, the argument used in the proof of decidability leads to an algorithm that has exponential worst-case complexity in the size n of a repository R. One possible algorithm to find *all* strong dependencies in a repository R is as follows.

Require: $R \neq \emptyset$
  `strongdeps` $\leftarrow \emptyset$
  for all $p, q \in R$ **do**
    if `strong_dependency`(p, q, R) **then**
      `strongdeps` $\leftarrow$ `strongdeps` $\cup \{p, q\}$
    end if
  end for
  return `strongdeps`

Where the function `strong_dependency` uses a SAT solver to check whether it is possible to install p without installing q (in repository R). This algorithm requires checking n^2 SAT instances, which is unfeasible with $n \approxeq 22,000$. We

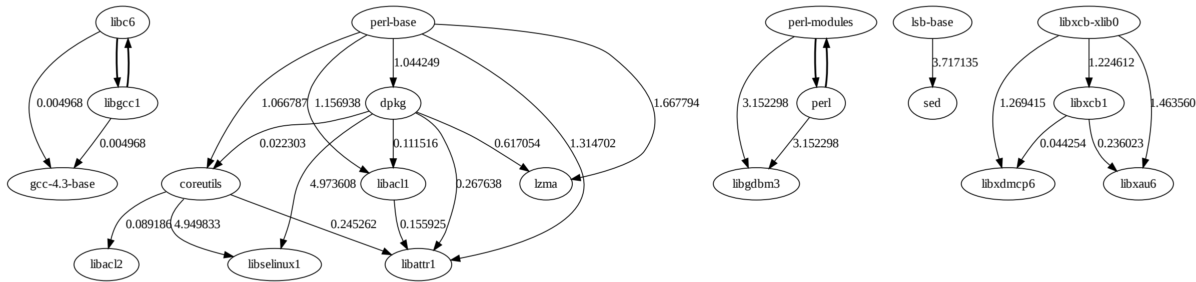


Figure 4. Dominance relations among the topmost 20 sensitive packages

need to look for an optimised approach; the following remark is the key observation.

Remark 4.2 (Reducing the search space)**.** *All packages q on which a given package p strongly depends are included in* any *installation of p. Furthermore, if a package p conjunctively depends on a package q, then q is a strong dependency of p.*

This leads to the following improved algorithm that strongly relies on the notion of installation sets and the property of transitivity of strong dependencies.

for all $p \in R$ **do**
  strongdeps $\leftarrow$ strongdeps $\cup$ conj_deps(p, R)
end for
for all $p \in R$ **do**
  $S \leftarrow$ install(p, R)
  for all $q \in S$ **do**
    if $(p, q) \notin$ strongdeps $\wedge$ strong_dep(p, q, R) **then**
      strongdeps $\leftarrow$ strongdeps $\cup \{p, q\}$
    end if
  end for
end for
return strongdeps

The function conj_deps(q, R) returns all packages in R that are connected to q, considering only conjunctive paths. We add to the strongdeps set all couples (p, q) such that there exists a conjunctive path between p and q, and then for all remaining packages in the install set of p, we check if there is a strong dependency using the SAT solver.

On one hand, the analysis of the structure of the repositories shows that it is in practise possible to find installation sets that are quite small. Considering only the installation set for a given package drastically reduces the number of calls to the SAT solver. On the other hand, since the large majority of strong dependencies can be derived directly from conjunctive dependencies, building the graph of conjunctive dependencies beforehand can further reduce the computation time.

In our experiments, calculating the strong dependency graph and sensitivity index for about $22,000$ packages takes about 5 minutes on a modern commodity Unix workstation.[7]

5. Perspective applications

The given notions of strong dependency, impact set, sensitivity, and strong dominance can be used to address issues showing up in the maintenance of large component repositories. In particular, we have identified two areas of application: repository-wide Quality Assurance (QA) and upgrade risk evaluation for user machines.

Quality Assurance FOSS distribution the size of Debian are not easily inspectable by hand, without specific tools. The work of release managers in such scenario is about maintaining a coherent package repository, i.e., in which each package is installable in at least one healthy installation. Such repositories are usually not built from scratch, but rather evolve from an unstable state to a stable one which is periodically released as the new major release of the distribution. Day to day maintenance of the repository includes actions such as adding packages to the repository (e.g., newly packaged software, or new releases) as well as removing them (e.g., superseded softwares or sub-standard quality packages which are not considered suitable for releasing). Quality assurance is meant to spot repository-wide incompatibilities or sub-standard quality packages, according to various criteria.

In such ecosystems, removing a package can have non-local effects which are not evident by just looking at the direct dependencies of the involved packages. For instance, removing a package p such that several packages depends on $p \mid q$ might be appropriate only if q is installable in

[7] Intel Xeon 3 GHz processor, 3 Gb of memory

the archive. The strong dependency graph can be used to detect similar cases efficiently. Once the graph has been computed—and Section 4 showed that the cost is affordable even for large distributions—detecting if a package is removable in isolation reduces to check whether its node has inbound edges or not. If really needed, following inbound edges can help building sets of packages removable as a whole.

In the same context, sensitivity can be used to decide when to freeze packages during the release process (decision currently delegated to folklore): the higher the sensitivity, the sooner a package should be frozen. Sensitivity can also be used to activate heuristic warnings in archive management tools when apparently innocuous packages are acted upon: attempting to remove or otherwise alter `gcc-4.3-base` at the end of the Lenny release process (see Table 2) would have surely been an error, in spite of the few packages mentioning it directly in their dependencies.

Upgrade risk evaluation System administrators of machines running FOSS distributions would like to be able to judge the risks of a certain upgrade. Risk evaluation not necessarily in the sense of deciding whether or not to perform an upgrade—not performing one is often not an option, due to the frequent case of upgrades that fix security vulnerability. Upgrade risk evaluation is nevertheless important to allocate suitable time slots to deploy upgrade plans proposed by package managers: the riskier the upgrade, the longer the time slot that should be planned for it.

The general principle we propose is that a package that is not strongly depended upon by other packages is relatively safe to upgrade; conversely, a package that is needed by many packages on the system might need some safety measures in case of problems (backup servers, ...). However this measure should be computed in relation to the actual user installation and not as an absolute value with respect to the distribution such as plain impact sets. Once the strong dependency graph of a user installation has been computed, the legacy package manager can be used to find upgrade plans as usual. On that plan the overall upgrade sensitivity can then be computed by summing up the size of the *installation impact sets* of all packages touched by the proposed plan; where the installation impact set of a package p is defined as the intersection of the strong impact set with the local installation.

The strong dependency graph used for risk evaluation must be the one corresponding to the distribution snapshot which was known *before* planning the upgrade. This is because we want to evaluate the risks with respect to the current installation, not to a future potential one in which package sensitivity can have changed. The maintenance of such graph on user machines is straightforward and can be postponed to after upgrade runs have been completed, in order to be ready for future upgrades.

Note that in this way, what is computed is an under approximation of the upgrade risk measure. For example consider the following scenario: a package p having **Depends:** `q | r`, and a healthy installation $I = \{p, q\}$. The direct dependencies of p entail no strong dependency, but in the given installation q has been "chosen" to solve p dependencies. Even if $p \notin Is(q, R) \cap I$, an upgrade of q in that specific installation has potentially an impact on p. The under approximation is nevertheless sound—i.e., all packages in the installation impact set are installed.

Release upgrades A particular case of upgrade are the so called *release upgrades* (or distribution upgrades) which are performed periodically to switch from an older stable release of a given distribution to a newer one. The relevance of such upgrades is that they usually affect almost all of the packages present in user installation. Such kind of upgrades are usually already performed wisely by system administrators devoting to them large time slots.

During release upgrades system administrators can be faced with the choice of whether to switch to a new major version of some available software or to stay with an older, legacy one. For instance, one can have the choice to switch to the Apache Web server 2.x series, or to stay with Apache 1.x. The upgrade is not forced by strict package versioning by either offering packages with different names (e.g. `apache1` vs `apache2` in Debian and its derivatives) or by avoiding explicit conflicts among the two set of versions (as it happens in RPM-based distributions). The choice is currently not technically well assisted: if `apache2` is tentatively chosen, the package manager will propose to upgrade all involved packages to the most recent version without highlighting which upgrades are *mandatory* to fulfil dependencies and which are not.

While this is a deficiency of state of the art solving algorithms [22], strong dependencies offer a cheap technical device to work around the problem with current solvers. It is enough to compute the strong dependency graph of both distributions and, in particular, the strong dependencies of the two (or more) involved packages. Then, by taking the difference of the strong dependencies in the new and in the old graph, the list of package which must be forcibly upgraded to do the switch is obtained. All such *forced upgrades* can then be presented to the administrator to better guide her or his choice.

6. Related works

Several interesting works have dealt with issues related to the topics touched by this paper. In the area of complex

networks, [14, 16] used FOSS distributions as case studies. The former is the closest to our focus, as it studies the network structure obtained from Debian inter-package relationships, showing that it is small-world, as the node connectivity follows a near power-law distribution. However, the analysis is performed on the direct dependency graph which, as discussed, misses the semantics of dependencies.

We could not get more information on how the data of [14] has been computed, as the snapshot of Debian used there comes from late 2004, and is no longer available in the Debian archives; based on the figures presented in the paper, and our analysis of the closest Debian stable distribution, we conclude that their analysis dropped all information about `Conflicts` and `Pre-Depends`. As a consequence, the figures produced for what is called in the paper "the 20 most highly depended upon packages" falls extremely short of reality: `libc6` is crucial for 3 times more packages than what is reported, and other critical packages such as `gcc-4.3-base` are entirely missed.

In the area of quality assurance for large software projects, many authors correlate component dependencies and past failure rates in order to predict future failures [24, 18, 19]. The underlying hypothesis is that software "fault-proneness" of a component is correlated to changes in components that are tightly related to it. In particular if a component A has many dependencies on a component B and the latter changes a lot between versions, one might expect that errors propagates through the network reducing the reliability of A. A related interesting statistical model to predict failures over time is the "weighted time damp model" that correlates most recent changes to software fault-proneness [9]. Social network methods [10] were also used to validate and predict the list of *sensitive* components in the Windows platform [24].

Our work differs for two main reasons. First, the source of dependency information is quite different. While dependency analysing for software components is inferred from the source code, the dependency information in software distributions are formally declared and can be assumed to be, on the average, trustworthy as reviewed by the package maintainer. Second, FOSS distributions still lack the needed data to correlate upgrade disasters with dependencies and hence to create statistical models that allow to predict future upgrade disasters. In more detail, the FOSS ecosystem is really fond of public bug tracker systems, but generally lacks explicit logging of upgrade attempts and a way to associate specific bugs to them. One of the goal of the Mancoosi[8] project—in which the authors are involved—is to create a corpus of upgrade problems which will be a first step in this direction.

The key idea behind the notion of sensitivity can be seen as a direct application of the evaluation of "disease spreading speed" in small world networks [23]: the higher the sensitivity, the larger the impact sets, the higher the (potential) bug spreading speed. The semantic definition of impact sets is crucial in this analysis: using the direct dependency graph would give no guarantee about which components will be effectively installed and therefore help bug spreading.

7. Conclusion and future work

This paper has introduced the novel notions of *strong dependencies* between software components, and of *sensitivity* as a measure of how many other components rely on the availability of a specific components; *strong dominance* has been introduced as well as a criterion to order and group components with similar sensitivity into meaningful clusters. We have shown concretely on a large scale real world example that such notions are better suited to describe true inter-component relationships than previous studies, which were solely based on the analysis of the syntactic (or direct) dependency graph. The main applications of these new notions are tools for quality assurance in large component ecosystems and upgrade risk evaluation.

The new notions have been tested on one of the largest known component-based system: Debian GNU/Linux, a popular FOSS distribution. Historical analysis of Debian strong and direct dependency graphs have been performed. Empirical evidence shows that, while the two notions are generally correlated, there are several components on which they give huge differences, with direct dependencies entirely missing key components that are correctly pinpointed by strong dependencies. We believe the case shown in this paper is strong enough to totally dismiss, in the future, measures built on direct dependencies as soon as the dependency language is expressive enough to encompass propositional logics.

We hence strongly advocate the evaluating of sensitivity on top of strong dependencies, and we have shown clearly how clustering components according to the notion of strong dominance allows to build a meaningful presentation of data, and uncover deep relationships among components in a repository.

Despite the theoretical complexity of the problem, and the sheer size of modern component repositories, we have succeeded in designing a simple optimised algorithm for computing strong dependencies that performs very well on real world instances, making all the measures proposed in this paper not only meaningful, but actually feasible.

Previous studies on network properties—such as small world characteristics—have been redone on the Debian strong dependency graph, showing that it stays small world.

Future works is planned in various directions. First of all the notion of installation impact set needs to be refined. While it is clear that the strong impact set is an under ap-

[8] `http://www.mancoosi.org`

proximation of it, it is less clear how to further refine it. On one hand we want to get closer to the actual set of potentially affected packages on a given machine. On the other it is not clear, for a package p depending on $q \mid r$ to which extent *both* packages should be considered as potentially impacted by a bug in p. It appears to be a limitation in the expressiveness of the dependency language which does not state an order between q and r, but needs further investigation. Interestingly enough, the implicit syntactic order "p before q" is already taken into account by some distribution tools such as build daemons and is hence worth modelling.

Distributions like Debian use a staged release strategy, in which two repositories are maintained: an "unstable" and a "testing" one. Packages get uploaded to unstable and migrate to testing when they satisfy some quality assurance criteria, including the goal of maintaining testing devoid of uninstallable packages. Current modelling of the problem is scarce and implementations rely on empirical package-by-package, brute force migration attempts. We believe that the notion of strong dependency and the clusters entailed by strong dominance can help in identifying clusters of packages which should forcibly migrate together.

Acknowledgements The authors would like to thank Yacine Boufkhad, Ralf Treinen, and Jerôme Vouillon for many interesting discussions on these issues.

References

[1] R. Albert, H. Jeong, and A. Barabasi. The diameter of the world wide web. *Nature*, 401:130–131, July 1999.

[2] R. Albert, H. Jeong, and A. Barabasi. Error and attack tolerance of complex networks. *Nature*, 406:378, 2000.

[3] Apache Software Foundation. Maven project. `http://maven.apache.org/`, 2009.

[4] E. Clayberg and D. Rubel. *Eclipse Plug-ins*. Addison-Wesley Professional, 3 edition, Dec. 2008.

[5] R. Di Cosmo, B. Durak, X. Leroy, F. Mancinelli, and J. Vouillon. Maintaining large software distributions: new challenges from the FOSS era. In *FRCSS 2006*, 2006. EASST Newsletter.

[6] R. Di Cosmo, P. Trezentos, and S. Zacchiroli. Package upgrades in FOSS distributions: Details and challenges. In *HotSWup'08*, 2008.

[7] S. Dick, A. Meeks, M. Last, H. Bunke, and A. Kandel. Data mining in software metrics databases. *Fuzzy Sets and Systems*, 145(1):81–110, 2004.

[8] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach, Revised*. Course Technology, 2 edition, Feb. 1998.

[9] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, 2000.

[10] R. A. Hanneman and M. Riddle. *Introduction to social network methods*. University of California, Riverside, 2005.

[11] I. Herraiz, G. Robles, R. Capilla, and J. Gonzalez-Barahona. Managing libre software distributions under a product line approach. In *COMPSAC'08*, pages 1221–1225, 2008.

[12] I. Jackson and C. Schwarz. Debian policy manual. `http://www.debian.org/doc/debian-policy/`, 2009.

[13] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Data mining for predictors of software quality. *International Journal of Software Engineering and Knowledge Engineering*, 9(5):547–564, 1999.

[14] N. LaBelle and E. Wallingford. Inter-package dependency networks in open-source software. *CoRR*, cs.SE/0411096, 2004.

[15] B. Livshits. Dynamine: Finding common error patterns by mining software revision histories. In *In ESEC/FSE*, pages 296–305. ACM Press, 2005.

[16] T. Maillart, D. Sornette, S. Spaeth, and G. V. Krogh. Empirical tests of zipf's law mechanism in open source linux distribution. *0807.0014*, June 2008.

[17] F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *ASE*, pages 199–208, 2006.

[18] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *ESEM*, pages 364–373, 2007.

[19] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *ACM Conference on Computer and Communications Security*, pages 529–540, 2007.

[20] G. Robles, J. M. Gonzalez-Barahona, M. Michlmayr, and J. J. Amor. Mining large software compilations over time: another perspective of software evolution. In *MSR '06*, pages 3–9. ACM, 2006.

[21] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley Professional, 1997.

[22] R. Treinen and S. Zacchiroli. Solving package dependencies: from EDOS to Mancoosi. In *DebConf 8: 9th conference of the Debian project*, 2008.

[23] D. J. Watts and S. H. Strogatz. Collective dynamics of small-world networks. *Nature*, 393(6684):440–442, June 1998.

[24] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *ICSE'08*, pages 531–540. ACM, 2008.

CHAPTER 10

Aeolus: a Component Model for the Cloud

This chapter contains the full text of the article
"Aeolus: a Component Model for the Cloud" *[52].*

Aeolus: a Component Model for the Cloud☆

Roberto Di Cosmo[a,b], Jacopo Mauro[b,c], Stefano Zacchiroli[a], Gianluigi Zavattaro[b,c]

[a] *Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126 CNRS, F-75205 Paris, France*
[b] *INRIA - Institut national de recherche en informatique et en automatique, France*
[c] *Department of Computer Science and Engineering, University of Bologna, Via Mura Anteo Zamboni 7, 40127 Bologna, Italy*

Abstract

We introduce the Aeolus component model, which is specifically designed to capture realistic scenarii arising when configuring and deploying distributed applications in the so-called *cloud* environments, where interconnected components can be deployed on clusters of heterogeneous virtual machines, which can be in turn created, destroyed, and connected on-the-fly.

The full Aeolus model is able to describe several component characteristics such as dependencies, conflicts, non-functional requirements (replication requests and load limits), as well as the fact that component interfaces to the world might vary depending on the internal component state.

When the number of components needed to build an application grows, it becomes important to be able to *automate* activities such as deployment and reconfiguration. This correspond, at the level of the model, to the ability to decide whether a desired target system configuration is reachable, which we call the *achievability* problem, and producing a path to reach it.

In this work we show that the achievability problem is undecidable for the full Aeolus model, a strong limiting result for automated configuration in the cloud. We also show that the problem becomes decidable, but Ackermann-hard, as soon as one drops non-functional requirements. Finally, we provide a polynomial time algorithm for the further restriction of the model where support for inter-component conflicts is also removed.

Keywords: software component, model, cloud computing, distributed systems

---

☆This paper is an extended and revised version of [1, 2] —a detailed comparison with these papers is reported in the related work Section 5. This work has been developed within the ANR-2010-SEGI-013-01 project Aeolus "Mastering the Complexity of the Cloud": we would like to thank all the project partners for the numerous discussions that contributed to the definition of the Aeolus model. This work has been partially performed at IRILL, center for Free Software Research and Innovation in Paris, France, http://www.irill.org

Email addresses: roberto@dicosmo.org (Roberto Di Cosmo), jmauro@cs.unibo.it (Jacopo Mauro), zack@pps.univ-paris-diderot.fr (Stefano Zacchiroli), zavattar@cs.unibo.it (Gianluigi Zavattaro)

1. Introduction

The expression "*cloud computing*" is broadly used to refer to the possibility of building sophisticated distributed software applications that can be run, on-demand, on virtualised hardware infrastructure at a fraction of the cost which was necessary just a few years ago. Reaping all the benefits of cloud computing is not an easy task: even when the infrastructure costs fall dramatically, the complexity of designing and maintaining distributed scalable software systems is a serious challenge.

Attempts are being made both in industry and in the research world to model and tame such complexity. On the industry side, a wealth of initiatives offer different kinds of solutions for isolated aspects of the problem. Tools like Puppet [3] or Chef [4] allow to automate the configuration of software components, based on a set of descriptions stored in a central server. CloudFoundry [5] allows to select, connect, and push to a cloud some predefined services (databases, message buses, proxies, . . .), that can be used as building blocks for writing applications using one of the supported frameworks. Finally, Juju [6] tries to extend the basic concepts of package managers—used by software distributions to automate software upgrades.

On the academic side, several teams are working, with different approaches, on the problems posed by the complexity of designing cloud applications. The Fractal component model [7], which itself pre-dates the popularization of the "cloud computing" expression, focuses on expressivity and flexibility: it provides a general notion of component assembly that can be used to describe concisely, and independently of the programming language, complex software systems. Building on Fractal, FraSCAti [8] provides a middleware that can be used to deploy applications in the cloud. ConfSolve [9] on the other hand aims at helping the application designer with some of the decisions to be made, and more specifically to optimally allocate virtual machines to concrete servers.

In all the above mentioned approaches, the goal is to allow the user (i.e., the application designer) to assemble a working system out of components that have been specifically designed or adapted to work together. The actual component selection (which web server should I use? which SQL database? which load balancer?) and interconnection (which front-end should I connect to which back-end, in order to avoid bottlenecks?) are the responsibility of the user. And if some reconfiguration needs to happen, it is either obtained by reassembling the system manually, or by writing specific code that is left for the user to write.

We believe that to make further progress in taming the complexity of sophisticated cloud applications, two major concerns must be taken into account.

Expressivity. We need component models that are expressive enough to capture all the component characteristics that are relevant for designing distributed, scalable applications which are typical in the cloud. Some of those characteristics (see Section 2 for a more in-depth discussion) are:

dependencies e.g., which other components should be deployed in order to be able to install, activate, upgrade, etc. a given component?

conflicts e.g., which other components, if any, would inhibit the deployment of a given component?

non-functional requirements e.g., if a component depends on others, how many of those would be needed to guarantee the desired level of fault-tolerance and/or load-balancing? Similarly, if a component offers functionalities to other, how many of them it can reasonably satisfy before needing to be replicated?

statefulness distributed/cloud-components have complex activation protocols, making their contextual requirements (dependencies, conflicts, etc.) vary over time, e.g., it might be enough to install a given component to be able to install another one, but the requirements to activate them might be different

Automation. While *expressivity* is certainly important, solving the challenge of designing and maintaining a cloud also requires automation. When the number of components grows, or the need to reconfigure appears more frequently, it is essential to be able to specify at a certain level of abstraction a particular target configuration of the distributed software system we want to realize, and to develop tools that provide a set of possible evolution paths leading from the current system configuration to one that corresponds to such a user request.

Automated approaches have been developed already, but thus far mostly for the particular case of configuring *package-based* FOSS (Free and Open Source Software) distributions on a *single* system, and there are generic, solver-based component managers for this task [10]. Similar approaches have been developed in the context of Software Product Lines where a correct instance of a product needs to be composed of a consistent set of features [11].

The goal of this paper is to lay the formal foundations of such an automated approach for the much more complex situation that arises when one needs to: (re-)configure not a single machine, but a variety of possibly "elastic" clusters of heterogeneous machines, living in different domains and offering interconnected services that need to be stopped, modified, and restarted in a specific order for the reconfiguration to be successful.

Contributions. We first elicit the expressivity requirements of a component model that is suitable for the cloud from specific use cases presented in Section 2. We then detail a formal component model for the cloud, called *Aeolus*, where components describe resources which provide and require different functionalities, and may be created or destroyed. As a major improvement over state-of-the-art component models, Aeolus components are equipped with *state machines* that *declaratively* describe how required and provided functionalities are enacted. The declarative information is essential to provide a planner with the input needed for exploring the possible evolution paths of the system, and propose a reconfiguration plan, which is the key automation enabler.

In Section 4 we study formally the complexity of checking the existence of a deployment plan in Aeolus, a property which we call *achievability*. We study

achievability in the full Aeolus model, as well as in more limited variants of it that exhibit different decidability and complexity characteristics.

We show that achievability is *undecidable* if one allows to impose capacity constraints—i.e., restrictions on the number of connections between required and provided functionalities—as it happens in the complete version of our model. This limiting result is particularly significant, as some industrial tools are starting to incorporate such restrictions to account for capacity limitations of services in the cloud.

If we remove the possibility of constraining the number of provided and required functionalities, we show that achievability becomes *decidable but Ackermann-hard*. Thus even in this simplified model, that we call *Aeolus core*, finding a plan can be extremely costly and infeasible from the computational point of view.

For this reason we consider a further restricted model, called Aeolus$^-$, where we drop the ability of stating capacity constraints on the provided and required functionalities, and declaring conflicts between resources. We prove that in Aeolus$^-$ achievability is *decidable in polynomial time*. This is interesting since Aeolus$^-$ corresponds to what mainstream industry tools can handle at present. Our result explains why it is still possible, in simple cases, to manage such systems manually.

2. A gentle introduction to Aeolus

We introduce the key features of Aeolus by eliciting them, step-by-step, from the analysis of realistic scenarii. As a running example, we consider several deployment use cases for WordPress, a popular weblog solution that requires several software services to operate, the main ones being a Web server and a SQL database. We present the use cases in order of increasing complexity ranging from the simplest ones, where everything runs on a single physical machine, to more complex ones where the whole appliance runs on a cloud.

Use case 1 — Package installation

Before considering the services that a machine is offering to others (locally or over the network), we need to model the *software installation* on the machine itself, so we will see how to model the three main components needed by WordPress, as far as their installation is concerned.

Software is often distributed according to the *package* paradigm [12], popularized by FOSS distributions, where software is shipped at the granularity of bundles called *packages*. Each package contains the actual software artifact, its default configuration, as well as a bunch of package metadata.

On a given machine, a software package may exists in different states (e.g., installed or uninstalled) and it should go through a complex sequence of states in different phases of unpacking and configuration to get there. In each of its states, similarly to what happens in most software component models [13], a package may have contextual *requirements* and offer some features, that we call

```
Package: wordpress
Version: 3.0.5+dfsg-0+squeeze1
Depends: httpd, mysql-client, php5, php5-mysql,
  libphp-phpmailer (>= 1.73-4), [...]

Package: mysql-server-5.5
Source: mysql-5.5
Version: 5.5.17-4
Provides: mysql-server, virtual-mysql-server
Depends: libc6 (>= 2.12), zlib1g (>= 1:1.1.4), debconf, [...]

Package: apache2
Version: 2.4.1-2
Maintainer: Debian Apache Maintainers <debian-apache@...>
Depends: lsb-base, procps, perl, mime-support,
  apache2-bin (= 2.4.1-2), apache2-data (= 2.4.1-2)
Conflicts: apache2.2-common
Provides: httpd
Description: Apache HTTP Server
```

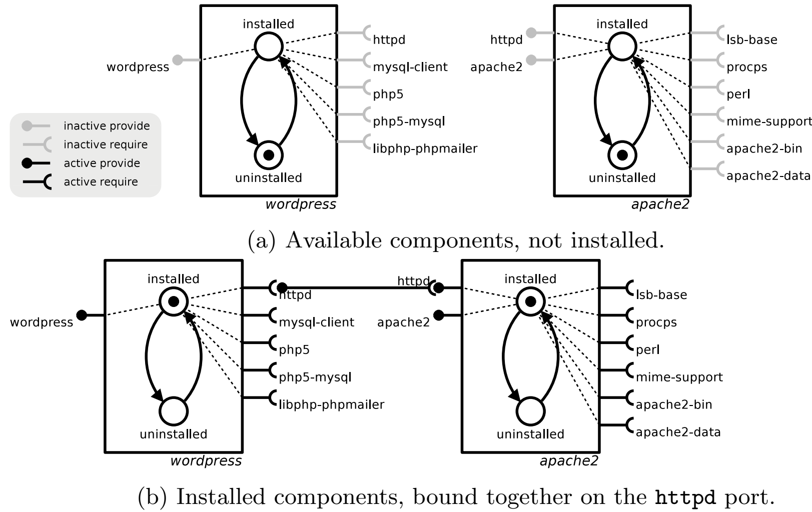
Figure 1: Debian package metadata for WordPress, Mysql and the Apache web server (excerpt)

provides. For instance in Debian, a popular FOSS distribution, there are packages for WordPress, Apache2 and MySQL equipped with metadata (reported in Figure 1) including a list of requirements (the *Depends* field) and of functionalities that are offered (the *Provides* field).

To model a software package, at this level of abstraction, we may use a simple state machine to capture its life cycle, with requirements and provides associated to each state. The ingredients of this model are very simple: a set of states Q, an initial state q_0, a transition function T from states to states, a set $\mathbf{R}$ of requirements, a set $\mathbf{P}$ of provides, and a function D that maps states to the requirements and provides that are *active* at that state. We call *component type* any such tuple $\langle Q, q_0, T, \langle \mathbf{P}, \mathbf{R} \rangle, D \rangle$, which will be formalized in Definition 1.

A system *configuration* is then built out of a collection of components that are instances of component types, with its current state, and a set of connections between requirements and provides of the different components. Connections indicate which provide is fulfilling the need of each requirement. A configuration is *correct* if all the requires which are active are satisfied by active provides; this will be made precise in Definition 4.

A straightforward graphical notation can capture all these pieces of information together: Figure 2 presents systems built using the components from Figure 1 (only modelling the dependency on `httpd` underlined in the metadata, for the sake of conciseness). In Figure 2a the packages are available but not installed yet. In Figure 2b the WordPress package is in the installed state and activates the requirement on `httpd`; Apache2 is also in the installed state, so the `httpd` provide is active and is used to satisfy the requirement, fact which is visualized by the *binding* connecting together the two *ports* named `httpd`.



(a) Available components, not installed.

(b) Installed components, bound together on the `httpd` port.

Figure 2

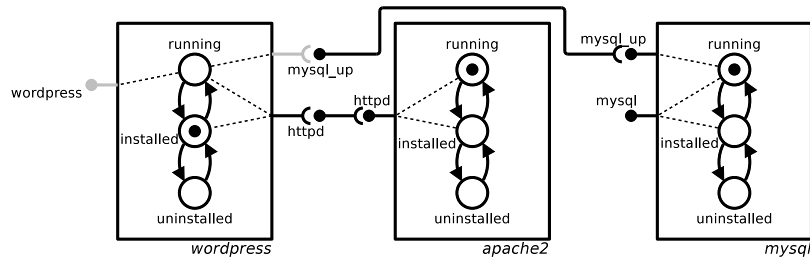


Figure 3: A graphical description of the basic model of services and packages.

Use case 2 — Services and packages

Installing the software on a single machine is a process that can already be automated using *package managers*: on Debian for instance, you only need to have an installed Apache server to be able to install WordPress. But bringing it *in production* requires to tune and activate the associated service, which is more tricky and less automated: the system administrator will need to edit configuration files so that WordPress knows the network addresses of an accessible MySQL instance.

The ingredients we have seen up to now in our model are sufficient to capture the dependencies among services, as shown in Figure 3. There we have added to each package an extra state corresponding to the activation of the associated service, and the requirement on `mysql_up` of the `running` state of WordPress captures the fact that WordPress cannot be started before MySQL is running. In this case, the bindings really correspond to a piece of configuration information,

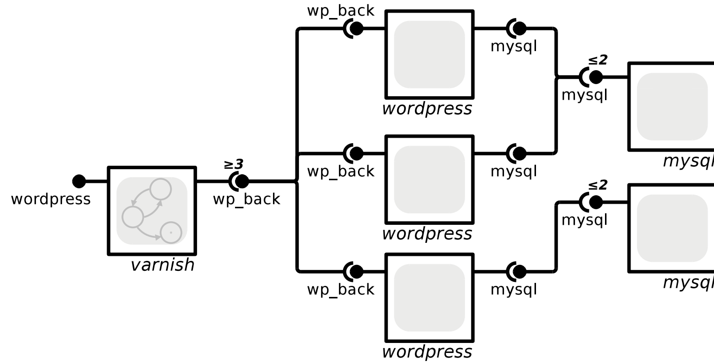


Figure 4: A graphical description of the model with redundancy and capacity constraints (internal sate machines are omitted for simplicity).

i.e., where to find a suitable MySQL instance.

Notice how this model does not impose any particular way of modelling the relations between packages and services. Instead of using a single component with an installed and a running state, we can simply model services and packages as different components, and relate them through dependencies.

Use case 3 — Redundancy, capacity planning, and conflicts

Services often need to be deployed on different machines to reduce the risk of failure or to increase the load they can withstand by the means of load-balancing. To properly design such scalable architectures system administrators might want, for instance, to indicate that a MySQL instance can only support a certain number of connected WordPress instances. Symmetrically, a WordPress hosting service may want to expose a reverse web proxy/load balancer to the public and require to have a minimum number of *distinct* instances of WordPress available as its back-ends.

To model this kind of situations, we allow capacity information to be added on provides and requires of each component in Aeolus: a number n on a provide port indicates that it can fulfil no more than n requirements, while a number n on a require port means that it needs to be connected to at least n provides from n *different* components.

As an example, Figure 4 shows the modelling of a WordPress hosting scenario where we want to offer high availability by putting the Varnish reverse proxy/load balancer in front of several WordPress instances, all connected to a cluster of MySQL databases.[1] For a configuration to be correct, the model

[1]All WordPress instances run within distinct Apache instances, which have been omitted for simplicity.

requires that Varnish is connected to at least 3 (active and distinct) WordPress back-ends, and that each MySQL instance does not serve more than 2 clients.

As a particular case, a 0 constraint on a require means that no provide with the same name can be active at the same time; this can be effectively used to model *global conflicts* between components. For instance, we can use this feature to model the conflict between the `apache2` and `apache2.2-common` packages that had been omitted in Figure 2.

Use case 4 — Creating and destroying components

Use cases like WordPress hosting are commonplace in the cloud, to the point that they are often used to showcase the capabilities of state-of-the-art cloud deployment technologies. The features of the model presented up to here are already expressive enough to encode these *static* deployment scenarii, where the system architecture does not evolve over time in reaction to load changes.

To model faithfully deployment runs on the cloud, where an arbitrary number of instances of virtual machine images can be allocated and deallocated on the fly, we also allow in our model creation and destruction of all kinds of components, provided they belong to some existing component type. For instance, in the configuration of Figure 4, to respond to an increase in traffic load one will need to spawn 2 new WordPress instances, which in turn will require to create new MySQL instances, as the available MySQL-s are no longer enough to handle the load increase.

3. The Aeolus model

We now formalize the *Aeolus model*, implementing all the features elicited from the use cases discussed in the previous section.

Notation. We assume given the following disjoint sets: $\mathcal{I}$ for interfaces and $\mathcal{Z}$ for components. We use $\mathbb{N}$ to denote strictly positive natural numbers, $\mathbb{N}_\infty$ for $\mathbb{N} \cup \{\infty\}$, and $\mathbb{N}_0$ for $\mathbb{N} \cup \{0\}$.

We model components as finite state automata indicating all possible component states and state transitions. When a component changes state, the sets of ports it requires from/provide to other components will also change: intuitively, the component interface with the external world varies with its state. A provide port represents the possibility of furnishing a functionality having a given interface. Similarly, a require port represent the need of a functionality with a given interface.

Definition 1 (Component type)**.** *The set* Γ *of* component types *of the Aeolus model, ranged over by* $\mathcal{T}_1, \mathcal{T}_2, \ldots$ *contains 5-ple* $\langle Q, q_0, T, P, D\rangle$ *where:*

- Q *is a finite set of states;*
- $q_0 \in Q$ *is the initial state and* $T \subseteq Q \times Q$ *is the set of* transitions*;*

- $P = \langle \mathbf{P}, \mathbf{R} \rangle$, *with* $\mathbf{P}, \mathbf{R} \subseteq \mathcal{I}$, *is a pair composed of the set of* provide *and the set of* require *ports, respectively;*
- D *is a function from* Q *to 2-ple in* $(\mathbf{P} \nrightarrow \mathbb{N}_\infty) \times (\mathbf{R} \nrightarrow \mathbb{N}_0)$.

Given a state $q \in Q$, $D(q)$ returns two partial functions $(\mathbf{P} \nrightarrow \mathbb{N}_\infty)$ and $(\mathbf{R} \nrightarrow \mathbb{N}_0)$ that indicate respectively the provide and require ports that q activates. The functions associate to the activate ports a numerical constraint indicating:

- for provide ports, the *maximum* number of bindings the port can satisfy,
- for require ports, the *minimum* number of required bindings to *distinct* components,
  - as a special case: if the number is 0 this indicates a conflict, meaning that there should be no other active port, in any other component, with the same name.

When the numerical constraint is not explicitly indicated, we assume as default value ∞ for provide ports (i.e., they can satisfy an unlimited amount of requires) and 1 for require (i.e., one provide is enough to satisfy the requirement). We also assume that the initial state q_0 has no demands (i.e., the second function of $D(q_0)$ has an empty domain).

Example 1. *Figure 2a depicts two component types:* wordpress *and* apache2*. In particular* wordpress *is formally defined as the 5-ple* $\langle Q, q_0, T, P, D \rangle$ *with:*

- $Q = \{\mathsf{uninstalled}, \mathsf{installed}\}$,
- $q_0 = \mathsf{uninstalled}$,
- $T = \{(\mathsf{uninstalled} \mapsto \mathsf{installed}), (\mathsf{installed} \mapsto \mathsf{uninstalled})\}$,
- $P = \langle \{\mathsf{wordpress}\},$
  $\{\mathsf{httpd}, \mathsf{mysql\text{-}client}, \mathsf{php5}, \mathsf{php5\text{-}mysql}, \mathsf{libphp\text{-}phpmailer}\} \rangle$,
- $D = \{(\mathsf{uninstalled} \mapsto \langle \emptyset, \emptyset \rangle),$
  $(\mathsf{installed} \mapsto \langle \{(\mathsf{wordpress} \mapsto \infty)\}, f \rangle)\}$
  where f *is a function that associates 1 to all require ports.*

We now define configurations that describe systems composed by component instances and bindings that interconnect them. A configuration, ranged over by $\mathcal{C}_1, \mathcal{C}_2, \ldots$, is given by a set of component types, a set of deployed components with a type and an actual state, and a set of bindings. Formally:

Definition 2 (Configuration)**.** *A* configuration $\mathcal{C}$ *is a 4-ple* $\langle U, Z, S, B \rangle$ *where:*

- $U \subseteq \Gamma$ *is the finite* universe *of all available component types;*
- $Z \subseteq \mathcal{Z}$ *is the set of the currently deployed* components*;*

- *S is the component* state description*, i.e., a function that associates to components in Z a pair $\langle \mathcal{T}, q \rangle$ where $\mathcal{T} \in U$ is a component type $\langle Q, q_0, T, P, D \rangle$, and $q \in Q$ is the current component state;*

- *$B \subseteq \mathcal{I} \times Z \times Z$ is the set of* bindings*, namely 3-ples composed by an interface, the component that requires that interface, and the component that provides it; we assume that the two components are distinct.*

Example 2. *Figure 2b depicts a configuration with two components and one binding. Formally, it corresponds to the 4-ple $\langle U, Z, S, B \rangle$ where:*

- *U is a set of component types including* wordpress *and* apache2*,*

- $Z = \{z_1, z_2\}$,

- $S = \{(z_1 \mapsto \langle \mathsf{wordpress}, \mathsf{installed} \rangle), (z_2 \mapsto \langle \mathsf{apache2}, \mathsf{installed} \rangle)\}$,

- $B = \langle \mathsf{httpd}, z_1, z_2 \rangle$.

In the following we will use a notion of configuration equivalence that relate configurations having the same instances up to renaming. This is used to abstract away from component identifiers and bindings.

Definition 3 (Configuration equivalence)**.** *Two configurations $\langle U, Z, S, B \rangle$ and $\langle U, Z', S', B' \rangle$ are equivalent, noted $\langle U, Z, S, B \rangle \equiv \langle U, Z', S', B' \rangle$, iff there exists a bijective function ρ from Z to Z' s.t.:*

1. *$S(z) = S'(\rho(z))$ for every $z \in Z$; and*
2. *$\langle r, z_1, z_2 \rangle \in B$ iff $\langle r, \rho(z_1), \rho(z_2) \rangle \in B'$.*

Notation. We write $\mathcal{C}[z]$ as a lookup operation that retrieves the pair $\langle \mathcal{T}, q \rangle = S(z)$, where $\mathcal{C} = \langle U, Z, S, B \rangle$. On such a pair we then use the postfix projection operators `.type` and `.state` to retrieve $\mathcal{T}$ and q, respectively. Similarly, given a component type $\langle Q, q_0, T, \langle \mathbf{P}, \mathbf{R} \rangle, D \rangle$, we use projections to (recursively) decompose it: `.states`, `.init`, and `.trans` return the first three elements; `.prov`, `.req` return $\mathbf{P}$ and $\mathbf{R}$; $.\mathbf{P}(q)$ and $.\mathbf{R}(q)$ return the two elements of the $D(q)$ tuple. When there is no ambiguity we take the liberty to apply the component type projections to $\langle \mathcal{T}, q \rangle$ pairs.

For example, $\mathcal{C}[z].\mathbf{R}(q)$ stands for the partial function indicating the active require ports (and their arities) of component z in configuration $\mathcal{C}$ when it is in state q.

We are now ready to formalize the notion of configuration correctness:

Definition 4 (Configuration correctness)**.** *Let us consider the configuration $\mathcal{C} = \langle U, Z, S, B \rangle$.*

We write $\mathcal{C} \models_{req} (z, r, n)$ to indicate that the require port of component z, with interface r, and associated number n is satisfied. Formally, if $n = 0$ all components other than z cannot have an active provide port with interface r,

namely for each $z' \in Z \setminus \{z\}$ such that $\mathcal{C}[z'] = \langle \mathcal{T}', q' \rangle$ we have that r is not in the domain of $\mathcal{T}'.\mathbf{P}(q')$. If $n > 0$ then the port is bound to at least n active ports, i.e., there exist n distinct components $z_1, \ldots, z_n \in Z \setminus \{z\}$ such that for every $1 \leq i \leq n$ we have that $\langle r, z, z_i \rangle \in B$, $\mathcal{C}[z_i] = \langle \mathcal{T}^i, q^i \rangle$ and r is in the domain of $\mathcal{T}^i.\mathbf{P}(q^i)$.

Similarly for provides, we write $\mathcal{C} \models_{prov} (z, p, n)$ to indicate that the provide port of component z, with interface p, and associated number n is not bound to more than n active ports. Formally, there exist no m distinct components $z_1, \ldots, z_m \in Z \setminus \{z\}$, with $m > n$, such that for every $1 \leq i \leq m$ we have that $\langle p, z_i, z \rangle \in B$, $S(z_i) = \langle \mathcal{T}^i, q^i \rangle$ and p is in the domain of $\mathcal{T}^i.\mathbf{R}(q^i)$.

The configuration $\mathcal{C}$ is correct *if for each component $z \in Z$, given $S(z) = \langle \mathcal{T}, q \rangle$ with $\mathcal{T} = \langle Q, q_0, T, P, D \rangle$ and $D(q) = \langle \mathcal{P}, \mathcal{R} \rangle$, we have that $(p \mapsto n_p) \in \mathcal{P}$ implies $\mathcal{C} \models_{prov} (z, p, n_p)$, and $(r \mapsto n_r) \in \mathcal{R}$ implies $\mathcal{C} \models_{req} (z, r, n_r)$.*

Example 3. *Figure 3* and *4 report examples of correct configurations. In Figure 3 it is easy to see that all active require ports are bound to an active provide port: this condition is enough when the numerical constraints has the default values.*

In Figure 4 there are two kinds of non-default numerical constraints: the constraint 3 on the require port wp_back *of the component of type* varnish *which is satisfied because there are at least three bindings connecting it to three distinct components (we assume that the* wp_back *provide ports of these three components are active) and the constraint 2 on the provide port* mysql *of the components of type* mysql *which are satisfied because those ports are connected to less than two bindings.*

We now formalize how configurations evolve from one state to another, by means of atomic actions:

Definition 5 (Actions)**.** *The set $\mathcal{A}$ contains the following actions:*

- *$stateChange(z, q_1, q_2)$ where $z \in \mathcal{Z}$;*
- *$bind(r, z_1, z_2)$ where $z_1, z_2 \in \mathcal{Z}$ and $r \in \mathcal{I}$;*
- *$unbind(r, z_1, z_2)$ where $z_1, z_2 \in \mathcal{Z}$ and $r \in \mathcal{I}$;*
- *$new(z : \mathcal{T})$ where $z \in \mathcal{Z}$ and $\mathcal{T}$ is a component type;*
- *$del(z)$ where $z \in \mathcal{Z}$.*

The execution of actions can now be formalized using a labelled transition systems on configurations, which uses actions as labels.

Definition 6 (Reconfigurations)**.** *Reconfigurations are denoted by transitions $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$ meaning that the execution of $\alpha \in \mathcal{A}$ on the configuration $\mathcal{C}$ produces a new configuration $\mathcal{C}'$. The transitions from a configuration $\mathcal{C} = \langle U, Z, S, B \rangle$ are defined as follows:*

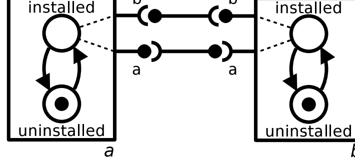


Figure 5: On the need of a *multiple state change*: how to install a and b?

$$\mathcal{C} \xrightarrow{stateChange(z,q_1,q_2)} \langle U, Z, S', B\rangle$$
$$\begin{array}{l} \textit{if } \mathcal{C}[z].\texttt{state} = q_1 \\ \textit{and } (q_1, q_2) \in \mathcal{C}[z].\texttt{trans} \\ \textit{and } S'(z') = \begin{cases} \langle \mathcal{C}[z].\texttt{type}, q_2\rangle & \textit{if } z' = z \\ \mathcal{C}[z'] & \textit{otherwise} \end{cases} \end{array}$$

$$\mathcal{C} \xrightarrow{bind(r,z_1,z_2)} \langle U, Z, S, B \cup \langle r, z_1, z_2\rangle\rangle$$
$$\begin{array}{l} \textit{if } \langle r, z_1, z_2\rangle \notin B \\ \textit{and } r \in \mathcal{C}[z_1].\texttt{req} \cap \mathcal{C}[z_2].\texttt{prov} \end{array}$$

$$\mathcal{C} \xrightarrow{unbind(r,z_1,z_2)} \langle U, Z, S, B \setminus \langle r, z_1, z_2\rangle\rangle \quad \textit{if } \langle r, z_1, z_2\rangle \in B$$

$$\mathcal{C} \xrightarrow{new(z:\mathcal{T})} \langle U, Z \cup \{z\}, S', B\rangle$$
$$\begin{array}{l} \textit{if } z \notin Z,\ \mathcal{T} \in U \\ \textit{and } S'(z') = \begin{cases} \langle \mathcal{T}, \mathcal{T}.\texttt{init}\rangle & \textit{if } z' = z \\ \mathcal{C}[z'] & \textit{otherwise} \end{cases} \end{array}$$

$$\mathcal{C} \xrightarrow{del(z)} \langle U, Z \setminus \{z\}, S', B'\rangle$$
$$\begin{array}{l} \textit{if } S'(z') = \begin{cases} \bot & \textit{if } z' = z \\ \mathcal{C}[z'] & \textit{otherwise} \end{cases} \\ \textit{and } B' = \{\langle r, z_1, z_2\rangle \in B \mid z \notin \{z_1, z_2\}\} \end{array}$$

Notice that in the definition of the transitions there is no requirement on the reached configuration: the correctness of these configurations will be considered at the level of deployment runs.

Also, we observe that there are configurations that cannot be reached through sequences of the actions we have introduced. In Figure 5, for instance, there is no way for package `a` and `b` to reach the installed state, as each package requires the other to be installed *first*. In practice, when confronted with such situations—that can be found for example in FOSS distributions in the presence

of loops of *Pre-Depends* that impose an order in the installation of two depending packages—current tools either perform all the state changes atomically, or abort deployment.

We want our planners to be able to propose deployment runs containing such atomic transitions. To this end, we introduce the notion of *multiple state change*:

Definition 7 (Multiple state change)**.** *A multiple state change $\mathcal{M} = \{stateChange(z^1, q_1^1, q_2^1), \cdots, stateChange(z^l, q_1^l, q_2^l)\}$ is a set of state change actions on different components (i.e., $z^i \neq z^j$ for every $1 \leq i < j \leq l$). We use $\langle U, Z, S, B\rangle \xrightarrow{\mathcal{M}} \langle U, Z, S', B\rangle$ to denote the effect of the simultaneous execution of the state changes in $\mathcal{M}$: formally,*

$$\langle U, Z, S, B\rangle \xrightarrow{stateChange(z^1,q_1^1,q_2^1)} \dots \xrightarrow{stateChange(z^l,q_1^l,q_2^l)} \langle U, Z, S', B\rangle$$

Notice that the order of execution of the state change actions does not matter as all the actions are executed on different components.

We can now define a *deployment run*, which is a sequence of actions that transform an initial configuration into a final correct one without violating correctness along the way. A deployment run is the output we expect from a planner, when it is asked how to reach a desired target configuration.

Definition 8 (Deployment run)**.** *A* deployment run *is a sequence $\alpha_1 \dots \alpha_m$ of actions and multiple state changes such that there exist $\mathcal{C}_i$ such that $\mathcal{C} = \mathcal{C}_0$, $\mathcal{C}_{j-1} \xrightarrow{\alpha_j} \mathcal{C}_j$ for every $j \in \{1, \dots, m\}$, and the following conditions hold:*

configuration correctness *for every $i \in \{0, \dots, m\}$, $\mathcal{C}_i$ is correct;*

multi state change minimality *if α_j is a multiple state change then there exists no proper subset $\mathcal{M} \subset \alpha_j$, or state change action $\alpha \in \alpha_j$, and correct configuration $\mathcal{C}'$ such that $\mathcal{C}_{j-1} \xrightarrow{\mathcal{M}} \mathcal{C}'$, or $\mathcal{C}_{j-1} \xrightarrow{\alpha} \mathcal{C}'$.*

Example 4. *Consider the configuration reported in Figure 3. Starting from an empty configuration. Such configuration can be reached upon execution of the following deployment run:*

$$\begin{array}{l} new(z_1 : \mathsf{wordpress}), \\ new(z_2 : \mathsf{apache2}), \\ stateChange(z_2, \mathsf{uninstalled}, \mathsf{installed}), \\ bind(\mathsf{httpd}, z_1, z_2), \\ stateChange(z_1, \mathsf{uninstalled}, \mathsf{installed}), \\ new(z_3 : \mathsf{mysql}), \\ stateChange(z_3, \mathsf{uninstalled}, \mathsf{installed}), \\ stateChange(z_3, \mathsf{installed}, \mathsf{running}), \\ bind(\mathsf{mysql_up}, z_1, z_3), \\ stateChange(z_2, \mathsf{installed}, \mathsf{running}), \end{array}$$

This sequence of actions is a deployment run because it guarantees the correctness of all the traversed configurations. Notice that this sequence of actions continues to be a deployment run even if $stateChange(z_1, \mathsf{uninstalled}, \mathsf{installed})$ *is postponed.*

On the contrary, it is no longer a deployment run if such action is anticipated because the requirement on the httpd *port is not yet fulfilled. It is no longer a deployment run even if such action is joined with other state changes to form a multiple state change action (like, e.g.,* $\{stateChange(z_1, \mathsf{uninstalled}, \mathsf{installed}), stateChange(z_2, \mathsf{installed}, \mathsf{running})\}$*) because this violates minimality.*

We now have all the ingredients to define the notion of *achievability*, that is our main concern: given a universe of component types, we want to know whether it is possible to deploy at least one component of a given component type $\mathcal{T}$ in a given state q.

Definition 9 (Achievability problem)**.** *The* achievability problem *has as input a universe U of component types, a component type $\mathcal{T}$, and a target state q. It returns as output **true** if there exists a deployment run $\alpha_1 \ldots \alpha_m$ such that $\langle U, \emptyset, \emptyset, \emptyset \rangle \xrightarrow{\alpha_1} \mathcal{C}_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_m} \mathcal{C}_m$ and $\mathcal{C}_m[z] = \langle \mathcal{T}, q \rangle$, for some component z in $\mathcal{C}_m$. Otherwise, it returns **false**.*

Example 5. *Consider the achievability problem for the universe of component types* wordpress*,* apache2*, and* mysql *in Figure 3, and the target expressed by* wordpress *in its* running *state. In this case the problem returns **true** because there exists, for instance, the deployment run obtained by adding* $stateChange(z_1, \mathsf{installed}, \mathsf{running})$ *at the end of the sequence of actions in Example 4.*

Notice that the restriction in this decision problem to one component in a given state is not limiting. One can easily encode any given final configuration by adding a dummy provide port enabled only by the required final states and a dummy component with requirements on all such provides.

4. Decidability and Complexity of Achievability

In this section, we establish our main results concerning the decidability and complexity of the achievability problem. The results change significantly depending on the restrictions imposed on the numerical constraints that are allowed as co-domains of the two $D(q)$ partial functions. We consider here three cases, which are detailed in the table below:

model	*co-domain*$(.\mathbf{P}())$	*co-domain*$(.\mathbf{R}())$
Aeolus$^-$	$\{\infty\}$	$\{1\}$
Aeolus core	$\{\infty\}$	$\{1, 0\}$
Aeolus	$\mathbb{N}_\infty$	$\mathbb{N}_0$

Aeolus (last row) is the same model of Definition 1, while Aeolus$^-$ is a restriction of it where only the default numerical constraints can be used: provide

ports always serve an unlimited amount of bindings, and require ports cannot conflict with other active ports, nor require a minimum number of bindings strictly higher than 1. Aeolus core, instead, is similar to Aeolus$^-$ but with the added possibility of expressing conflicts.

In the following we will show that: achievability is undecidable in Aeolus; it is decidable, but not primitive recursive (i.e., Ackermann-hard) in Aeolus core; it is decidable and polynomial in Aeolus$^-$.

4.1. Achievability is undecidable in Aeolus

The proof that achievability is undecidable is by reduction from the reachability problem in 2 Counter Machines (2CMs) [14], a well-known Turing-complete computational model.

A 2CM is a machine with *two registers* R_1 and R_2 holding arbitrary large natural numbers and a *program* P consisting of a finite sequence of numbered instructions of the two following types:

- $j : \mathsf{Inc}(R_i)$: increments R_i and goes to the instruction $j + 1$;
- $j : \mathsf{DecJump}(R_i, l)$: if the content of R_i is not zero, then decreases it by 1 and goes to the instruction $j + 1$, otherwise jumps to the l instruction.

A state of the machine is given by a tuple (i, v_1, v_2) where i indicates the next instruction to execute (the program counter) and v_1 and v_2 are the values contained in the two registers, respectively.

Notation. In the following we use the notation $(i, v_1, v_2) \rightarrow (i', v_1', v_2')$ to say that the state of the machine changes from (i, v_1, v_2) to (i', v_1', v_2') as effect of the execution of the i-th instruction.

It is not restrictive to assume that the initial configuration of the machine is $(1, 0, 0)$. In 2CMs, the problem of checking whether a given l-th instruction is reachable from the initial configuration is undecidable.

We model a 2CM as follows. We use a component to simulate the execution of the program instructions. The contents v_i of the register R_i is modelled by v_i components in a particular state r_i. Increment instructions add one component in this state r_i, while decrement instructions move one component in state r_i to a different state. The state r_i activates a provide port one_i, so the simulation of a test for zero has simply to check the absence in the environment of active one_i ports.

The component types used to model 2CMs in Aeolus are depicted in Figure 6. Namely, we consider four component types: $\mathcal{T}_P$ to simulate the execution of the program instructions, $\mathcal{T}_{R_1}$ and $\mathcal{T}_{R_2}$ for the two registers and $\mathcal{T}_B$ used to guarantee that the components involved in the simulation cannot be deleted.

In $\mathcal{T}_P$ we assume one state q_j for each instruction j. If the j-th instruction is $j : \mathsf{Inc}(R_i)$ (see the state q_j in Figure 6), a protocol with three intermediary states is executed that completes by entering the state q_{j+1}, representing the next instruction to execute. This protocol has the effect to force a component

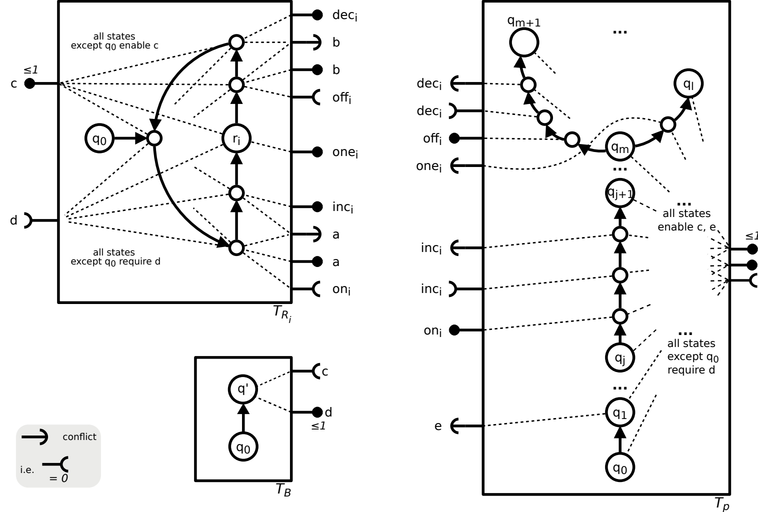


Figure 6: Modeling 2 counter machines (2CMs) in the Aeolus model.

of type $\mathcal{T}_{R_i}$ to execute a complementary protocol that completes by entering the state r_i, thus representing the increment by one of the register R_i. A description of this protocol is reported in the proof of Proposition 1. If the m-th instruction is $m : \mathsf{DecJump}(R_i, l)$ (see the state q_m in Figure 6), two state changes are possible from the state q_m. The first one starts a protocol similar to the previous one, whose effect here is to force one component of type $\mathcal{T}_{R_i}$ to exit from the state r_i, thus representing the decrement by one of the register R_i. The second one traverses a state that requires the absence in the configuration of active one_i provide port, thus checking that the content of R_i is zero, and then enters state q_l.

In our model, when a component z is not used to satisfy requirements, it could be removed by executing the $del(z)$ action. The cancellation of a component of type $\mathcal{T}_{R_i}$ could then erroneously change register contents during the simulation. To avoid that, we force the connection of each component of type $\mathcal{T}_{R_i}$ with a corresponding instance of a component of type $\mathcal{T}_B$. These types of components reciprocally connect through the ports c and d as soon as they move from their initial state q_0. Such connections remain active during the entire simulation, ensuring that components will not be deleted by mistake. Notice that it is necessary to add the capacity constraint 1 to the provide ports c and d, in order to have an exact one-to-one correspondence between the components of type $\mathcal{T}_{R_i}$ and those of type $\mathcal{T}_B$.

As a final remark, notice that the first state q_1 of the component type $\mathcal{T}_P$ has a requirement on the absence in the environment of an active provide port e, port which is activated by all the states in $\mathcal{T}_P$. This guarantees that at most one component of type $\mathcal{T}_P$ can be in a state different from q_0. Moreover, we also have to avoid that such component is removed by a del action: this can

be guaranteed by using the same pairing technique with a component of type $\mathcal{T}_B$ described above. It is sufficient to impose that all the states of $\mathcal{T}_P$, but q_0, activate a provide port on c with numerical constraint 1, and a require port on d, as shown in Figure 6.

We are now ready to formally prove our undecidability result. In the following we assume given a 2CM program P and use $\mathcal{C}^{\#}_{\langle \mathcal{T},q\rangle}$ to denote the number of components of type $\mathcal{T}$ in state q in the configuration $\mathcal{C}$.

Definition 10. *Let (i, v_1, v_2) be a state of a 2CM. We define*

$$\begin{aligned} \mathcal{C}_0 &= \langle\{\mathcal{T}_P, \mathcal{T}_{R_1}, \mathcal{T}_{R_2}, \mathcal{T}_B\}, \emptyset, \emptyset, \emptyset\rangle \\ [\![(i, v_1, v_2)]\!] &= \{\ \mathcal{C} \ \mid \ \mathcal{C} \text{ is a correct conf. with universe } \{\mathcal{T}_P, \mathcal{T}_{R_1}, \mathcal{T}_{R_2}, \mathcal{T}_B\}, \\ &\qquad \mathcal{C}^{\#}_{\langle \mathcal{T}_P,q_i\rangle} = 1,\ \mathcal{C}^{\#}_{\langle \mathcal{T}_{R_1},r_1\rangle} = v_1,\ \text{and } \mathcal{C}^{\#}_{\langle \mathcal{T}_{R_2},r_2\rangle} = v_2\ \} \end{aligned}$$

In the following we call *program step* a sequence of reconfigurations that, beyond other actions, includes state changes of the component $\mathcal{T}_P$ until entering a state q_j (corresponding to an instruction of the program P). Formally, it is a non empty sequence of reconfigurations $\mathcal{C}_1 \xrightarrow{\alpha_1} \mathcal{C}_2 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{m-1}} \mathcal{C}_m$ such that:

- there exists an index j of a program instruction[2] for which ${\mathcal{C}_m}^{\#}_{\langle \mathcal{T}_P,q_j\rangle} = 1$ while ${\mathcal{C}_{m-1}}^{\#}_{\langle \mathcal{T}_P,q_j\rangle} = 0$;
- for every $1 < i < m$ there exists no index j of a program instruction for which ${\mathcal{C}_i}^{\#}_{\langle \mathcal{T}_P,q_j\rangle} = 1$ while ${\mathcal{C}_{i-1}}^{\#}_{\langle \mathcal{T}_P,q_j\rangle} = 0$.

Notice that in our modeling of 2CMs there exists also infinite sequences of reconfigurations that do not contain program steps: in these cases they include infinitely many actions that are irrelevant for the simulation (like creation or destruction of components, or bindings and unbindings) and only finitely many state changes of components of type $\mathcal{T}_P$ that are not sufficient to reach a new q_j state.

We first observe that the deployment run composed by the actions $new(z_1 : \mathcal{T}_P)$, $new(z_2 : \mathcal{T}_B)$, $bind(c, z_2, z_1)$, $bind(d, z_1, z_2)$, and the multi stage change action $\{stateChange(z_1, q_0, q_1), stateChange(z_2, q_0, q')\}$ guarantees the possibility to reach, from the initial empty configuration $\mathcal{C}_0$, a configuration corresponding to the initial state of the 2CM, i.e., a configuration in $[\![(1, 0, 0)]\!]$. Moreover, every *program step* from $\mathcal{C}_0$ reaches a configuration in $[\![(1, 0, 0)]\!]$. In fact, it is not possible for components of type $\mathcal{T}_{R_i}$ to enter their state r_i if components of type $\mathcal{T}_P$ perform only the state change action from q_0 to q_1.

Fact 1. *There exists a deployment run from $\mathcal{C}_0$ to a configuration in $[\![(1, 0, 0)]\!]$. Moreover, for every* program step *from $\mathcal{C}_0$ to a configuration $\mathcal{C}'$, we have that $\mathcal{C}' \in [\![(1, 0, 0)]\!]$.*

[2]Notice that 0 is not a correct index as we have assumed that the program P starts from instruction 1.

The proof of undecidability is based on two distinct propositions, a first one about *completeness* of the simulation (i.e., each computational step of the 2CM can be mimicked by a deployment run), and a second one about *soundness* (i.e., each program step of a configuration $\mathcal{C} \in [\![(j, v_1, v_2)]\!]$ corresponds to a step $(j, v_1, v_2) \rightarrow (j', v_1', v_2')$ of the 2CM).

Proposition 1. *Let* (j, v_1, v_2) *be a state of the 2CM and let* $\mathcal{C} \in [\![(j, v_1, v_2)]\!]$. *If* $(j, v_1, v_2) \rightarrow (j', v_1', v_2')$ *then there exists a deployment run from* $\mathcal{C}$ *to a configuration* $\mathcal{C}' \in [\![(j', v_1', v_2')]\!]$.

Proof. It is sufficient to perform an analysis of the three possible computational steps of the 2CM: increment, decrement and test for zero. We detail only the increment case (the other cases are treated similarly). If the j-th instruction is an increment on R_i then in $\mathcal{C}$ the component of type $\mathcal{T}_P$ is in state q_j. This means that an action can be executed to move it in the state that activates the on_i provide port (see Figure 6). This permits to create a new pair of components of type $\mathcal{T}_{R_i}$ and $\mathcal{T}_B$, bind them on their ports c and d, and then move the former in the state requiring on_i (notice that a multiple state change is needed to satisfy the mutual requirements between the two new components). The deployment run can then be extended by moving the new component of type $\mathcal{T}_{R_i}$ in the state that activates the provide port inc_i, moving the component of type $\mathcal{T}_P$ in state q_{j+1} (with two state changes) and finally the new component of type $\mathcal{T}_{R_i}$ in its state r_i. The reached configuration $\mathcal{C}'$ belongs to $[\![(j', v_1', v_2')]\!]$ because $\mathcal{C}'^{\#}_{\langle \mathcal{T}_{R_i}, r_i \rangle} = \mathcal{C}^{\#}_{\langle \mathcal{T}_{R_i}, r_i \rangle} + 1$ and in this case $j' = j + 1$. □

We now move to the proof of the soundness result.

Proposition 2. *Let* (j, v_1, v_2) *be a state of the 2CM and let* $\mathcal{C} \in [\![(j, v_1, v_2)]\!]$. *If there exists a program step from* $\mathcal{C}$ *that reaches a configuration* $\mathcal{C}'$ *then* $\mathcal{C}' \in [\![(j', v_1', v_2')]\!]$ *and* $(j, v_1, v_2) \rightarrow (j', v_1', v_2')$.

Proof. We perform an analysis of the reconfiguration actions executed during the program step. There are three kinds of actions: state changes of the component of type $\mathcal{T}_P$ moving from state q_j to $q_{j'}$, state changes inside one of the components $\mathcal{T}_{R_i}$ and other actions. The other actions can be creation or destruction of resources, creation or deletion of bindings (that do not alter the configuration correctness), and multi state changes of new pairs of components of type $\mathcal{T}_{R_i}$ and $\mathcal{T}_B$. All these actions are irrelevant as their modifications on the configuration have no impact on the properties checked by the definition of $[\![(j', v_1', v_2')]\!]$. It is now sufficient to perform a case analysis on the three possible kinds of state changes from state q_j to $q_{j'}$ in the component of type $\mathcal{T}_P$: increment, decrement, and test for zero.

In the "test for zero" case, we have that the j-instruction is of the kind $\mathsf{DecJump}(R_i, j')$. Moreover, in the configuration (j, v_1, v_2) we have $v_i = 0$ because during the program step no component of type $\mathcal{T}_{R_1}$ or $\mathcal{T}_{R_2}$ can perform state changes (this would require the activation of either the port on_i or the port off_i) and the component of type $\mathcal{T}_P$ traverses a state that checks

the absence of active one_i ports (this implies $\mathcal{C}^{\#}_{\langle \mathcal{T}_{R_i}, r_i \rangle} = 0$). Hence, we have $(j, v_1, v_2) \to (j', v_1, v_2)$ and $\mathcal{C}' \in [\![(j', v_1, v_2)]\!]$.

In the other two cases, it is sufficient to check that the execution of a protocol like the one described in the proof of Proposition 1 is executed by the component of type $\mathcal{T}_P$ and one component of type $\mathcal{T}_{R_i}$. □

We can finally state the main undecidability result:

Theorem 1. *The achievability problem is undecidable in the Aeolus model.*

Proof. Let M be a 2CM with program P, and let $U = \{\mathcal{T}_P, \mathcal{T}_{R_1}, \mathcal{T}_{R_2}, \mathcal{T}_B\}$ be the set of the corresponding component types defined as in Figure 6.

We have that $(1, 0, 0) \to^* (j, v_1, v_2)$ *if and only if* there exists a deployment run from $\mathcal{C}_0$ to a configuration $\mathcal{C} \in [\![(j, v_1, v_2)]\!]$. The *only if* part follows from Fact 1 and the Proposition 1, while the *if* part follows from Fact 1 and the Proposition 2. Hence we have that the j-instruction is reachable in M *if and only if* the achievability problem is satisfied for the universe U, the component type $\mathcal{T}_P$ and the state q_j.

The undecidability of achievability thus follows from the undecidability of reachability for 2CMs. □

4.2. Achievability is decidable in Aeolus core

We demonstrate decidability of the achievability problem by resorting to the theory of Well-Structured Transition Systems (WSTS) [15, 16].

A reflexive and transitive relation is called *quasi-ordering*. A *well-quasi-ordering* (wqo) is a quasi-ordering $(X, \leq)$ such that, for every infinite sequence $x_1, x_2, x_3, \cdots$, there exist $i < j$ with $x_i \leq x_j$. Given a quasi-order $\leq$ over X, an *upward-closed set* is a subset $I \subseteq X$ such that the following holds: $\forall x, y \in X : (x \in I \wedge x \leq y) \Rightarrow y \in I$. Given $x \in X$, its upward closure is $\uparrow x = \{y \in X \mid x \leq y\}$. This notion can be extended to sets in the obvious way: given a set $Y \subseteq X$ we define its upward closure as $\uparrow Y = \bigcup_{y \in Y} \uparrow y$. A *finite basis* of an upward-closed set I is a finite set B such that $I = \bigcup_{x \in B} \uparrow x$.

Definition 11. *A WSTS is a transition system $(\mathcal{S}, \to, \preceq)$ where $\preceq$ is a wqo on $\mathcal{S}$ which is* compatible *with $\to$, i.e., for every $s_1 \preceq s_1'$ such that $s_1 \to s_2$, there exists $s_1' \to^* s_2'$ such that $s_2 \preceq s_2'$ ($\to^*$ is the reflexive and transitive closure of $\to$). Given a state $s \in \mathcal{S}$, $Pred(s)$ is the set $\{s' \in \mathcal{S} \mid s' \to s\}$ of immediate predecessors of s. $Pred$ is extended to sets in the obvious way: $Pred(S) = \bigcup_{s \in S} Pred(s)$. A WSTS has* effective pred-basis *if there exists an algorithm that, given $s \in \mathcal{S}$, returns a finite basis of $\uparrow Pred(\uparrow s)$.*

The following proposition is a special case of Proposition 3.5 in [16].

Proposition 3. *Let $(\mathcal{S}, \to, \preceq)$ be a finitely branching WSTS with decidable $\preceq$ and effective pred-basis. Let I be any upward-closed subset of $\mathcal{S}$ and let $Pred^*(I)$ be the set $\{s' \in S \mid s' \to^* s\}$ of predecessors of states in I. A finite basis of $Pred^*(I)$ is computable.*

In the remainder of this section, we assume a given universe U of component types; so we can consider that the set of distinct component type and state pairs $\langle \mathcal{T}, q \rangle$ is finite. Let k be its cardinality. We will resort to the theory of WSTS by considering an abstract model of configurations in which bindings are not taken into account.

Definition 12 (Abstract Configuration)**.** *An abstract configuration $\mathcal{B}$ is a finite multiset of pairs $\langle \mathcal{T}, q \rangle$ where $\mathcal{T}$ is a component type and q is a corresponding state. We use Conf to denote the set of abstract configurations.*

A concretization of an abstract configuration is simply a correct configuration that for every component type and state pair $\langle \mathcal{T}, q \rangle$ has as many instances of component $\mathcal{T}$ in state q as pairs $\langle \mathcal{T}, q \rangle$ in the abstract configuration.

Definition 13 (Concretization)**.** *Given an abstract configuration $\mathcal{B}$ we say that a correct configuration $\mathcal{C} = \langle U, Z, S, B \rangle$ is one concretization of $\mathcal{B}$ if there exists a bijection f from the multiset $\mathcal{B}$ to Z s.t. $\forall \langle \mathcal{T}, q \rangle \in \mathcal{B}$ we have that $S(f(\langle \mathcal{T}, q \rangle)) = \langle \mathcal{T}, q \rangle$. We denote with $\gamma(\mathcal{B})$ the set of concretizations of $\mathcal{B}$. We say that an abstract configuration $\mathcal{B}$ is correct if it has at least one concretization (formally $\gamma(\mathcal{B}) \neq \emptyset$).*

An interesting property of an abstract configuration is that from one of its concretizations it is possible to reach via bind and unbind actions all the other concretizations (up to instance renaming). This is because it is always possible to switch one binding from one provide port to another one by adding a binding to the new port and then removing the old binding.

Property 1. *Given an abstract configuration $\mathcal{B}$ and configurations $\mathcal{C}_1, \mathcal{C}_2 \in \gamma(\mathcal{B})$ there exists $\alpha_1, \cdots, \alpha_n$ sequence of binding and unbinding actions s.t. $\mathcal{C}_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} \mathcal{C} \equiv \mathcal{C}_2$.*

We now move to the definition of our quasi-ordering on abstract configurations. In order to be able to exploit the WSTS techniques in our context, we need to consider a quasi-ordering which is compatible with the notion of correctness, i.e., given a correct abstract configuration, all the greater configurations must be correct as well. For this reason, we cannot adopt the usual multiset inclusion ordering. In fact, the addition of one component to a correct configuration could introduce a conflict. If the type-state pair of the added component was absent in the configuration, the conflict might be with an already present component of a different type-state. If the type-state pair was present in a single copy, the new conflict might be with that component if the considered type-state pair activates one provide and one conflict port on the same interface. This sort of self-conflict is revealed when there are at least two instances, as one component cannot be in conflict with itself (by definition of correctness). If the type-state pair was already present in at least two copies, no new conflicts can be added otherwise such conflicts were already present in the configuration (thus contradicting its correctness).

In the light of the above observation, we define an ordering on configurations that corresponds to the product of three orderings: the identity on the set of

type-state pairs that are absent, the identity on the pairs that occurs in one instance, and the multiset inclusion for the projections on the remaining type-state pairs.

Definition 14 ($\leq$)**.** *Given a pair $\langle \mathcal{T}, q \rangle$ and an abstract configuration $\mathcal{B}$, let $\#_{\mathcal{B}}(\langle \mathcal{T}, q \rangle)$ be the number of occurrences in $\mathcal{B}$ of the pair $\langle \mathcal{T}, q \rangle$. Given two abstract configurations $\mathcal{B}_1, \mathcal{B}_2$ we write $\mathcal{B}_1 \leq \mathcal{B}_2$ if for every component type $\mathcal{T}$ and state q we have that $\#_{\mathcal{B}_1}(\langle \mathcal{T}, q \rangle) = \#_{\mathcal{B}_2}(\langle \mathcal{T}, q \rangle)$ when $\#_{\mathcal{B}_1}(\langle \mathcal{T}, q \rangle) \in \{0, 1\}$ or $\#_{\mathcal{B}_2}(\langle \mathcal{T}, q \rangle) \in \{0, 1\}$, and $\#_{\mathcal{B}_1}(\langle \mathcal{T}, q \rangle) \leq \#_{\mathcal{B}_2}(\langle \mathcal{T}, q \rangle)$ otherwise.*

As discussed above, this ordering is compatible with correctness.

Property 2. *If an abstract configuration $\mathcal{B}$ is correct than all the configurations $\mathcal{B}'$ such that $\mathcal{B} \leq \mathcal{B}'$ are also correct.*

Another interesting property of the $\leq$ quasi-ordering is that from one concretization of an abstract configuration, it is always possible to reconfigure it to reach a concretization of a smaller abstract configuration. In this case it is possible to first add from the starting configuration the bindings that are present in the final configuration. Then the extra components present in the starting configuration can be deleted because not needed to guarantee correctness (they are instances of components that remain available in at least two copies). Finally the remaining extra bindings can be removed.

Property 3. *Given two abstract configurations $\mathcal{B}_1, \mathcal{B}_2$ s.t. $\mathcal{B}_1 \leq \mathcal{B}_2$, $\mathcal{C}_1 \in \gamma(\mathcal{B}_1)$, and $\mathcal{C}_2 \in \gamma(\mathcal{B}_2)$ we have that there exists a deployment run $\mathcal{C}_2 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} \mathcal{C} \equiv \mathcal{C}_1$.*

We have that $\leq$ is a wqo on *Conf* because, as we consider finitely many component type-state pairs, the three distinct orderings that compose $\leq$ are themselves wqo.

Lemma 1. *$\leq$ is a wqo over Conf.*

Proof. The proof is based on a representation of abstract configurations as 3-ples of tuples: namely, given $\mathcal{B} \in Conf$ we represent it as the triple $\langle a, b, c \rangle$ where a is used to represent the component type-state pairs with cardinality 0 in $\mathcal{B}$, b represents those with cardinality 1, and c describes all the other pairs. We assume a total ordering on the set (of cardinality k) of the possible type-state pairs. The three elements a, b and c are vectors of arity k such that $a[i] = 1$ (resp. $b[i] = 1$) if the i-th component type-state pair has cardinality 0 (resp. 1) in $\mathcal{B}$ and $a[i] = 0$ (resp. $b[i] = 0$) otherwise, while $c[i]$ contains the cardinality of the i-th pair in $\mathcal{B}$ if it is greater or equal to 2 and $c[i] = 0$ otherwise. Consider now two abstract configurations $\mathcal{B}_1, \mathcal{B}_2 \in Conf$ and the corresponding triple representations $\langle a_1, b_1, c_1 \rangle$ and $\langle a_2, b_2, c_2 \rangle$. We have that $\mathcal{B}_1 \leq \mathcal{B}_2$ iff $a_1 = a_2$, $b_1 = b_2$ and $c_1 \leq^k c_2$ (where $\leq^k$ is the extension of the standard ordering on natural numbers to vectors of length k).

The equality on bit vectors of length k (a and b are indeed of length k) is a wqo as there are only finitely many such vectors (namely, 2^k). Dickson's

lemma [17] states that a product of wqo is a wqo, thus $\leq^k$ is a wqo too. We can conclude that the ordering on the triples is a wqo by applying again Dickson's lemma. □

We now define a transition system on abstract reconfigurations and prove it is a WSTS with respect to the ordering defined above.

Definition 15 (Abstract reconfigurations)**.** *We write $\mathcal{B} \rightarrow \mathcal{B}'$ if there exists $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$ for some $\mathcal{C} \in \gamma(\mathcal{B})$ and $\mathcal{C}' \in \gamma(\mathcal{B}')$.*

Lemma 2. *The transition system $(Conf, \rightarrow, \leq)$ is a WSTS.*

Proof. The $\leq$ is a wqo for *Conf* by Lemma 1. To prove the thesis we need to prove that $\leq$ is compatible with $\rightarrow$ (i.e., if $\mathcal{B}_1 \leq \mathcal{B}_2$ and $\mathcal{B}_1 \rightarrow \mathcal{B}_1'$ then $\mathcal{B}_2 \rightarrow^* \mathcal{B}_2'$ for some $\mathcal{B}_2'$ s.t. $\mathcal{B}_1' \leq \mathcal{B}_2'$). This is straightforward since we have $\mathcal{B}_2 \rightarrow^* \mathcal{B}_1$ (by Property 3), $\mathcal{B}_1 \rightarrow \mathcal{B}_1'$ (by hypothesis), and $\mathcal{B}_1' \leq \mathcal{B}_1'$ (by reflexivity of $\leq$). □

The following lemma is rather technical and it will be used to prove that $(Conf, \rightarrow, \leq)$ has effective pred-basis. Intuitively it will allow us to consider, in the computation of the predecessors, only finitely many different (multiple) state change actions.

Lemma 3. *Let k be the number of distinct component type-state pairs. If $\mathcal{B}_1 \rightarrow \mathcal{B}_2$ then there exists $\mathcal{B}_1' \rightarrow \mathcal{B}_2'$ such that $\mathcal{B}_1' \leq \mathcal{B}_1$, $\mathcal{B}_2' \leq \mathcal{B}_2$ and $|\mathcal{B}_2'| \leq 2k + 2k^2$.*

Proof. If $|\mathcal{B}_2| \leq 2k+2k^2$ the thesis trivially holds. Consider now $|\mathcal{B}_2| > 2k+2k^2$ and a transition $\mathcal{C}_1 \xrightarrow{\alpha} \mathcal{C}_2$ such that $\mathcal{C}_1 \in \gamma(\mathcal{B}_1)$ and $\mathcal{C}_2 \in \gamma(\mathcal{B}_2)$. We now show that is possible to remove one component from $\mathcal{C}_1$ while keeping the possibility to perform an action leading to a configuration corresponding to $\mathcal{C}_2$ without the component removed form $\mathcal{C}_1$. We consider two subcases.
Case 1. There are three components z_1, z_2 and z_3 having the same component type and internal state that do not perform a state change in the action α. Without loss of generality we can assume that z_3 does not appear in α (this is not restrictive because at most two components that do not perform a state change can occur in an action). We can now consider the configuration $\mathcal{C}_1'$ obtained by $\mathcal{C}_1$ after removing z_3 (if there are bindings connected to provide ports of z_3, these can be rebound to ports of z_1 or z_2). Consider now $\mathcal{C}_1' \xrightarrow{\alpha} \mathcal{C}_2'$ and the corresponding abstract configurations $\mathcal{B}_1'$ and $\mathcal{B}_2'$. We have that $\mathcal{B}_1' \rightarrow \mathcal{B}_2'$, $\mathcal{B}_1' \leq \mathcal{B}_1$, $\mathcal{B}_2' \leq \mathcal{B}_2$ and $|\mathcal{B}_2'| < |\mathcal{B}_2|$. If $|\mathcal{B}_2'| \leq 2k + 2k^2$ the thesis is proved, otherwise we repeat this deletion of components.
Case 2. There are no three components of the same type-state that do not perform a state change. Since $|\mathcal{B}_2| > 2k + 2k^2$ we have that α is a multiple state change involving strictly more than $2k^2$ components (otherwise there are strictly more than $2k$ components that do not perform state changes, thus at least three of them are of the same type-state). This ensures the existence of three components z_1, z_2 and z_3 of the same type that perform the same state change from q to q'. As in the previous case we consider the configuration $\mathcal{C}_1'$ obtained by $\mathcal{C}_1$ after removing z_3 and α' the state change similar to α but

without the state change of z_3. Consider now $\mathcal{C}'_1 \xrightarrow{\alpha'} \mathcal{C}'_2$ and the corresponding abstract configurations $\mathcal{B}'_1$ and $\mathcal{B}'_2$. As above, $\mathcal{B}'_1 \leq \mathcal{B}_1$, $\mathcal{B}'_2 \leq \mathcal{B}_2$ and $|\mathcal{B}'_2| < |\mathcal{B}_2|$. If $|\mathcal{B}'_2| \leq 2k + 2k^2$ the thesis is proved, otherwise we repeat the deletion of components. □

We are now in place to prove that $(Conf, \rightarrow, \leq)$ has effective pred-basis.

Lemma 4. *The transition system* $(Conf, \rightarrow, \leq)$ *has effective pred-basis.*

Proof. We first observe that given an abstract configuration the set of its concretizations up to configuration equivalence is finite, and that given a configuration $\mathcal{C}$ the set of preceding configurations $\mathcal{C}'$ such that $\mathcal{C}' \xrightarrow{\alpha} \mathcal{C}$ is also finite (and effectively computable). Consider now an abstract configuration $\mathcal{B}$. We now show how to compute a finite basis for $\uparrow Pred(\uparrow \mathcal{B})$ by considering the preceding configurations of a finite set of corresponding concrete configurations. First of all we consider the finite set of abstract configurations composed by $\mathcal{B}$, if $|\mathcal{B}| > 2k+2k^2$, or all the configurations $\mathcal{B}'$ such that $\mathcal{B} \leq \mathcal{B}'$ and $|\mathcal{B}'| \leq 2k+2k^2$, otherwise. Then we consider the (finite) set of concretizations of all such abstract configurations. Finally we compute the (finite) set of the preceding configurations of all such concretizations. The finite basis is obtained by taking the set of abstract configurations corresponding to the latter: this is finite and it is a basis for $\uparrow Pred(\uparrow \mathcal{B})$ as a consequence of Lemma 3. □

We are finally ready to prove our decidability result.

Theorem 2. *The achievability problem in Aeolus core is decidable.*

Proof. Let k be the number of distinct component type-state pairs according to the considered universe of component types. We first observe that if there exists a correct configuration containing a component of type $\mathcal{T}$ in state q then it is possible to obtain via some binding, unbinding, and delete actions another correct configuration with k or less components. Hence, given a component type $\mathcal{T}$ and a state q, the number of target configurations that need to be considered is finite. Moreover, given a configuration $\mathcal{C}' \in \gamma(\mathcal{B}')$ there exists a deployment run from $\mathcal{C} \in \gamma(\mathcal{B})$ to $\mathcal{C}'$ iff $\mathcal{B} \in Pred^*(\uparrow \mathcal{B}')$.

To solve the achievability problem it is therefore possible to consider only the (finite set of) abstractions of the target configurations. For each of them, say $\mathcal{B}'$, by Proposition 3, Lemma 2, and Lemma 4 we know that a finite basis for $Pred^*(\uparrow \mathcal{B}')$ can be computed. It is sufficient to check whether the initial empty configuration is in such basis. □

In this section we have considered just the problem of reaching a target configuration starting from an initial empty configuration. The proof presented holds however also for the more general problem of finding if the target configuration can be reached by an initial (possibly non empty) configuration. Indeed, in this case, it is sufficient to check whether at least one of the abstract configurations in $Pred^*(\uparrow \mathcal{B}')$ contains a configuration that is $\leq$ w.r.t. the abstraction of the initial configuration.

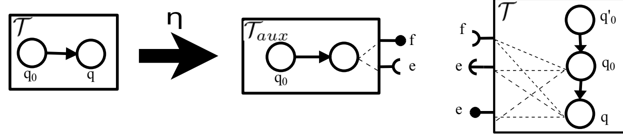


Figure 7: Example of a component type transformation η.

4.3. Achievability is Ackermann-hard in Aeolus core

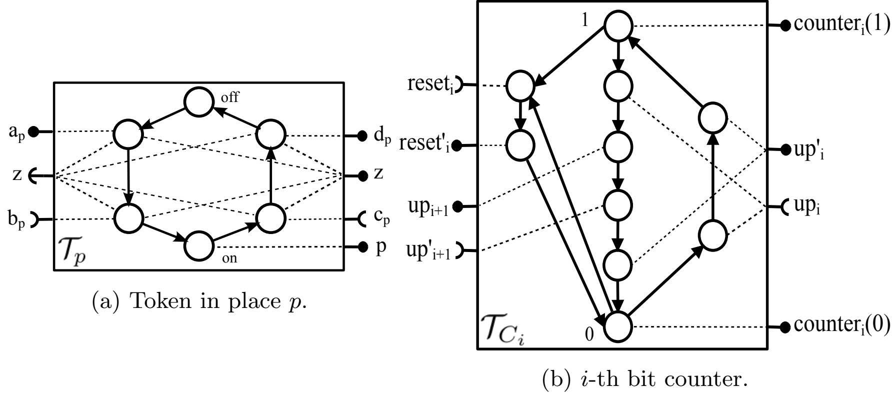
We now prove that the achievability problem in Aeolus core is Ackermann-hard by reduction from the coverability problem in reset Petri nets, a problem which is indeed known to be Ackermann-hard [18].

We start with some background on reset Petri nets.

A *reset Petri net RN* is a tuple $\langle P, T, \vec{m_0}\rangle$ such that P is a finite set of *places*, T is a finite set of *transitions*, and $\vec{m_0}$ is a marking, i.e., a mapping from P to $\mathbb{N}$ that defines the initial number of tokens in each place of the net. A transition $t \in T$ is defined by a mapping ${}^\bullet t$ (preset) from P to $\mathbb{N}$, a mapping $t^\bullet$ (postset), and by a set of reset arcs $t\!\downarrow \subseteq P$. A configuration is a marking $\vec{m}$. Transition t is enabled at marking $\vec{m}$ iff ${}^\bullet t(p) \leq \vec{m}(p)$ for each $p \in P$. Firing t at $\vec{m}$ leads to a new marking $\vec{m'}$ defined as $\vec{m'}(p) = \vec{m}(p) -^\bullet t(p) + t^\bullet(p)$ if $p \notin t\!\downarrow$, and $\vec{m'}(p) = 0$ otherwise; we denote this marking transformation with $\vec{m} \mapsto \vec{m'}$. A marking $\vec{m}$ is reachable from $\vec{m_0}$ if $\vec{m_0} \mapsto^* \vec{m}$, i.e., it is possible to produce $\vec{m}$ after firing finitely many times transitions in T. Given a reset net $\langle P, T, \vec{m_0}\rangle$ and a marking $\vec{m}$, the coverability problem consists in checking for the existence of a reachable marking $\vec{m'}$ such that $\vec{m} \leq \vec{m'}$, i.e. $\vec{m}(p) \leq \vec{m'}(p)$ for every $p \in P$. In [18] it is proved that the coverability problem for reset nets is Ackermann-hard.

Before entering into the details of our modelling of reset Petri nets, we observe that given a component type $\mathcal{T}$ it is always possible to modify it in such a way that its instances are persistent and unique. The uniqueness constraint can be enforced by allowing all the states of the component type to provide a new port with which they are in conflict. To avoid the component deletion it is sufficient to impose its reciprocal dependence with a new type of component. When this dependence is established the components cannot be deleted without violating it. In Figure 7 we show an example of how a component type having two states can be modified in order to reach our goal. A new auxiliary initial state q'_0 is created. The new port e ensures that the instances of type $\mathcal{T}$ in a state different from q'_0 are unique. The require port f provided by a new component type $\mathcal{T}_{aux}$ forbids the deletion of the instances of type $\mathcal{T}$, if they are not in state q'_0. We assume that the ports e and f are fresh. We can therefore consider, without loss of generality, components that are unique and persistent. Given a component type $\mathcal{T}$ we denote this component type transformation with $\eta(\mathcal{T})$.

We now consider a given reset Petri net $RN = \langle P, T, \vec{m_0}\rangle$ and discuss how to encode it in Aeolus core component types. We will use three types of com-



(a) Token in place p.

(b) i-th bit counter.

Figure 8: Token and counter component types.

ponents: one modelling the tokens, one for the transitions and one for defining a counter. The components for the transitions and the counter are unique and persistent, while those for the tokens cannot be unique because the number of tokens in a Petri net can be unbounded. The simplest component type, denoted with $\mathcal{T}_p$, is the one used to model a token in a given place $p \in P$. Namely, one token in a place p is encoded as one instance of $\mathcal{T}_p$ in the *on* state. There could be more than one of these components deployed simultaneously representing multiple tokens in a place. In Figure 8a we represent the component type $\mathcal{T}_p$. The initial state is the *off* state. The token could be created following a protocol consisting of requiring the port a_p and then providing the port b_p. Symmetrically, a token can be removed by providing the port c_p and then requiring the port d_p. Even if multiple instances of the token component can be deployed simultaneously, only one of them at a time can initiate the protocol to change its state. This is guaranteed by the conflict on the port z, which is provided by all the states of the state change protocols. The component provides the port p when it is in the *on* state.

In order to model the transitions without having an exponential blow up of the size of the encoding we need a mechanism to count up to a fixed number. Indeed, a transition can consume and produce a given number of tokens from and to several places. To count a number up to n we will use instances of the component types $\mathcal{T}_{C_1}, \cdots, \mathcal{T}_{C_{\lceil \log(n) \rceil}}$; the type $\mathcal{T}_{C_i}$ will be used to represent the i-th less significant bit of the binary representation of the counter that, for our purposes, needs just to support the increment and reset operations. In Figure 8b we represent one of the bits implementing the counter. The initial state is 0. To increment the bit it is necessary to provide and require in sequence the up_i and up_i' ports, while to reset it the $reset_i$ and $reset_i'$ ports. If the bit is in state 1 the increment will trigger the increment of the next bit (except for the component representing the most significant bit that will never need to do that). The instance of $\mathcal{T}_{C_i}$ can be used to count how many tokens are consumed

or produced by checking if the right number is reached via the ports *counter*$_i$*(0)* and *counter*$_i$*(1)*. We transform the component types $\mathcal{T}_{C_1}, \cdots, \mathcal{T}_{C_{\lceil \log(n) \rceil}}$ using the η transformation to ensure uniqueness and persistence of its instances.

The transitions in T can be represented with a single component interacting with token and counter components. This component, represented in Figure 9a, during a so-called initialization phase, performs state changes until reaching a state q. The initialization phase is used to generate the representation of the initial marking $\vec{m_0}$. From the state q it can nondeterministically select one transition t to fire, by entering a corresponding q_t state. The subsequent state changes can be divided in three phases: consumption, production, and reset. These phases respectively model the consumption of tokens from the places in the preset of the transition t, the production of tokens in the places in the postset of t, and the complete elimination of the tokens in the reset places of t. Notice that, as the transition t to be fired is selected nondeterministically, the corresponding deployment run could block due to the unavailability of instances of token components required during the consumption phase. As we will discuss later, these blocking deployment runs are not problematic.

We now describe in details the three consumption, production and reset phases, and then comment the initialization phase. In the consumption phase, for every place p in the preset of the transition, the counter is first reset providing the $reset_i$ and requiring the $reset'_i$ ports for all the counter bits. Then a cycle starts incrementing the counter, by providing and requiring the ports up_1 and up'_1, and consuming a token, by providing and requiring the ports c_p and d_p. The cycle ends when all the bits of the counter represent in binary the right number of tokens that need to be consumed from p. If instead at least one bit is wrong the cycle restarts. In Figure 9b we depict the part of the component type modelling the consumption of n tokens from the place p.

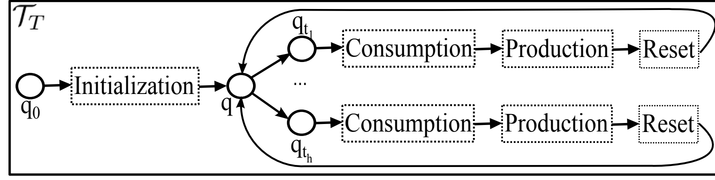
The production phase is similar to the consumption phase. For every kind of token that needs to be produced, the counter components are used to count the actual number of instances. The production of a single token follows a protocol similar to the one used for their consumption with the only difference that the ports a_p and b_p are required and provided in sequence, instead of providing and requiring c_p and d_p.

Reset arcs are instead modelled with a single state conflicting with the tokens in places that must be reset. For instance in Figure 9c we depicted the part of the component modelling the reset of a place p: the conflict on the port p forces the deletion of all the instances of component type $\mathcal{T}_p$ in the *on* state. At the end of the reset phase, the component has a transition to return in its state q.

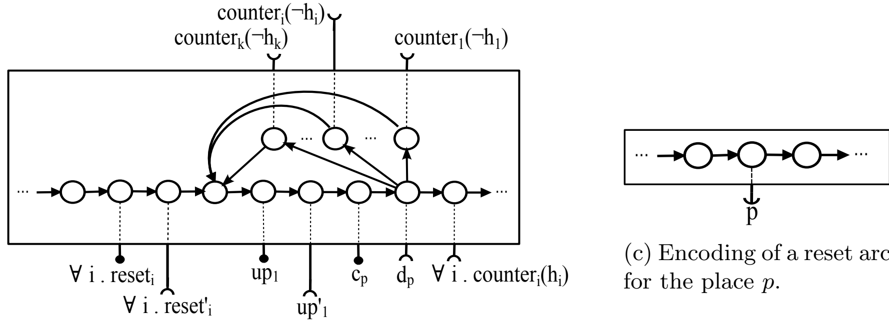
The initialization phase is like a production phase in which the tokens of $\vec{m_0}$ are produced; at the end of the initialization phase the state q is entered.

We will denote with $\mathcal{T}_T$ the component type explained above.

Definition 16 (Reset Petri net encoding in Aeolus core)**.** *Given a reset Petri net $RN = (P, T, m_0)$ if n is the largest number of tokens that can be consumed or produced by a transition in T, the encoding of RN in Aeolus core is the set*



(a) Transitions component.



(b) Consumption phase of n tokens from place p for a transition t ($k = \lceil \log(n) \rceil$ and h_i is the i-th least significative bit of the binary representation of n).

(c) Encoding of a reset arc for the place p.

Figure 9

of component types:

$$\Gamma_{RN} = \{\mathcal{T}_p \mid p \in P\} \cup \{\eta(\mathcal{T}_{C_i}) \mid i \in [1..\lceil \log(n) \rceil]\} \cup \{\eta(\mathcal{T}_T)\}$$

Notice that the components of type $\mathcal{T}_p$ are not guaranteed to be persistent, so they can be deleted even when they are in the *on* state. This corresponds to a nondeterministic elimination of one token from the place p. As we will discuss later, this token elimination in our net simulation is not problematic because it is not necessary to faithfully reproduce the net behaviour, but it is sufficient to preserve coverability (i.e., the possibility to generate at least a predefined number of tokens in some given places).

Before moving to the proof of the correspondence between the reset Petri net RN and its encoding Γ_{RN}, we observe that the size of the component types Γ_{RN} is polynomial w.r.t. the size of the reset Petri net. This is due to the fact that the counter and place components have a constant amount of states and ports while the component for the transitions has a number of states that grows linearly with respect to the number of places involved in the transitions.

We now introduce the notation $\mathcal{C}_0$ for denoting the empty initial configuration of our encoding, and $[\![\vec{m}]\!]$ to characterize configurations corresponding to the net marking $\vec{m}$.

Definition 17. *Let $RN = (P, T, m_0)$ be a reset Petri net and $\vec{m}$ one of its*

markings. We define:

$$\begin{array}{rcl} \mathcal{C}_0 & = & \langle \Gamma_{RN}, \emptyset, \emptyset, \emptyset \rangle \\ {[\vec{m}]} & = & \{ \ \mathcal{C} \ \mid \ \ \mathcal{C} \textit{ is a correct configuration with universe } \Gamma_{RN}, \\ & & \qquad \mathcal{C}^{\#}_{\langle \mathcal{T}_T, q \rangle} = 1, \quad \forall p \in P. \mathcal{C}^{\#}_{\langle \mathcal{T}_p, on \rangle} = \vec{m}(p) \ \} \end{array}$$

We call *net step* a sequence of reconfigurations on components instances of the universe Γ_{RN} that, beyond other actions, includes state changes of the component $\mathcal{T}_T$ until entering the state q. Formally, it is a non empty sequence of reconfigurations $\mathcal{C}_1 \xrightarrow{\alpha_1} \mathcal{C}_2 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{m-1}} \mathcal{C}_m$ such that ${\mathcal{C}_m}^{\#}_{\langle \mathcal{T}_T, q \rangle} = 1$, while ${\mathcal{C}_i}^{\#}_{\langle \mathcal{T}_T, q \rangle} = 0$, for every $1 < i < m$.

We first observe that the deployment run that creates an instance of $\mathcal{T}_T$ and then performs the state changes in the initialization phase until entering the state q guarantees the possibility to reach a configuration in $[\vec{m_0}]$. Moreover, every *net step* from the initial empty configuration reaches a configuration in $[\vec{m}]$ where $\vec{m} \leq \vec{m_0}$. Notice that we need to consider markings $[\vec{m}]$ smaller than $[\vec{m_0}]$ because token components moved in the *on* state during the initialization could be nondeterministically deleted.

Fact 2. *There exists a deployment run from $\mathcal{C}_0$ to a configuration in $[\vec{m_0}]$. Moreover, for every* net step *from $\mathcal{C}_0$ to a configuration $\mathcal{C}'$, we have that $\mathcal{C}' \in [\vec{m}]$ and $\vec{m} \leq \vec{m_0}$.*

The proof of correspondence between a reset Petri net RN and its encoding Γ_{RN} is based on two distinct propositions, a first one about *completeness* of the simulation (i.e., each firing of a net transition can be mimicked by a deployment run), and a second one about *soundness* (i.e., each net step of a configuration corresponds to the firing of a net transition).

Proposition 4. *Let $RN = (P, T, m_0)$ be a reset Petri net, $\vec{m}$ one of its markings, and $\mathcal{C}$ a configuration in $[\vec{m}]$. If $\vec{m} \mapsto \vec{m'}$ then there exists a deployment run from $\mathcal{C}$ to a configuration $\mathcal{C}' \in [\vec{m'}]$.*

Proof. It is sufficient to observe that if $\vec{m} \mapsto \vec{m'}$ then there exists a transition $t \in T$ that, by consuming and producing tokens and resetting places, transforms $\vec{m}$ in $\vec{m'}$. This transition can be selected in a deployment from $\mathcal{C}$ that starts by changing the state of $\mathcal{T}_T$ form q to q_t. Then the corresponding consumption, production and reset phases can be executed to reach a configuration in $[\vec{m'}]$. $\square$

We now move to the proof of the soundness result by showing that, if there exists a net step from a configuration in $[\vec{m}]$ to a configuration in $[\vec{m'}]$, then there exists a marking $\vec{m''} \geq \vec{m'}$ such that $\vec{m} \mapsto \vec{m''}$. We need to consider a greater marking in the net because, as already observed, token components could be deleted during the deployment run and a perfect correspondence between the reset Petri net and its simulation is not guaranteed.

Proposition 5. *Let $RN = (P, T, m_0)$ be a reset Petri net, $\vec{m}$ one of its markings, and $\mathcal{C}$ a configuration in $[\vec{m}]$ having a net step to $\mathcal{C}'$. Then, there exists a marking $\vec{m'}$ such that $\mathcal{C}' \in [\vec{m'}]$ and a marking $\vec{m''} \geq \vec{m'}$ such that $\vec{m} \mapsto \vec{m''}$.*

Proof. We first observe that the final configuration $\mathcal{C}'$ of a net step contains the $\mathcal{T}_T$ component in the q state; thus there exists $[\vec{m'}]$ s.t. $\mathcal{C}' \in [\vec{m'}]$.

The net step from $\mathcal{C} \in [\vec{m}]$ to $\mathcal{C}' \in [\vec{m'}]$ includes the state changes, on the instance of component type $\mathcal{T}_T$, corresponding to the consumption, production and reset phases for some transition t in T. The execution of the consumption phase guarantees that ${}^\bullet t \leq \vec{m}$ thus t can fire in $\vec{m}$: let $\vec{m} \mapsto \vec{m''}$ be the effect of firing such transition. We have that $\vec{m''} \geq \vec{m'}$ because $\vec{m''}$ is obtained from $\vec{m}$ by performing the same consumption, production and reset executed during the net step from $\mathcal{C} \in [\vec{m}]$ to $\mathcal{C}' \in [\vec{m'}]$. Notice that during this net step some token component in the *on* state could be nondeterministically deleted, but this has no impact on the property $\vec{m''} \geq \vec{m'}$ because its effect is simply to remove more instances of active token components w.r.t. those removed during the consumption phase. □

Notice that besides the net step from $\mathcal{C}$ to $\mathcal{C}'$ considered in the above Proposition, there are deployment runs starting from $\mathcal{C}$ that do not correspond to net steps. For instance, there could be an infinite sequence of creations and deletions of components or, more interesting, a non habilitated transition t could be tried. In this case the deployment run could block because the consumption phase cannot be completed. These additional deployment runs are not problematic as our encoding needs to preserve the possibility to reach specific target configurations (i.e., those that contain at least a given amount of instances of token components in the *on* state), and additional deployment runs that do not reach such configurations are irrelevant.

We are finally ready to prove Ackermann-hardness of the achievability problem for Aeolus core.

Theorem 3. *The achievability problem for Aeolus core is Ackermann-hard.*

Proof. Consider a reset Petri net $RN = \langle P, T, \vec{m_0} \rangle$ and a target marking $\vec{m}$. The problem of checking whether $\vec{m}$ can be covered in RN is Ackermann-hard [18]. We first construct a new reset Petri net $RN' = \langle P \uplus \{p'\}, T \uplus \{t'\}', \vec{m_0} \rangle$ with an additional place p' and a transition t' such that ${}^\bullet t' = m$ and $t'^\bullet = \{p'\}$. It is immediate to see that $\vec{m}$ can be covered in RN iff at least one token can be placed in p' in RN'. Moreover this transformation increases the size of the Petri net by a constant. We now consider the set of component types $\Gamma_{RN'}$, i.e., the encoding of RN' in Aeolus core. We have already observed that the size of $\Gamma_{RN'}$ is polynomial w.r.t. the size of the reset net RN'. We complete the proof by showing that a token can be placed in p' in RN' *iff* the component type-state pair $\langle \mathcal{T}_T, q_{p'} \rangle$ is achievable with the universe of component types $\Gamma_{RN'}$, where $q_{p'}$ is the state of $\mathcal{T}_T$ that provides the port $b_{p'}$, necessary to move in the *on* state a component of type $\mathcal{T}_{p'}$.

The “only if” part follows from Fact 2 and Proposition 4, that guarantee the existence of a deployment run reaching a configuration $\mathcal{C} \in [\![\vec{m}]\!]$ with $m(p') > 0$. As in $\mathcal{C}$ there is at least one instance of type $\mathcal{T}_{p'}$ in the *on* state, at least one configuration is traversed with the instance $\mathcal{T}_T$ in the $q_{p'}$ state.

The proof of the “if” part proceeds as follows. The achievability of the pair $\langle \mathcal{T}_T, q_{p'} \rangle$ guarantees the existence of a deployment run $\mathcal{C}_0 \xrightarrow{\alpha_1} \mathcal{C}_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} \mathcal{C}_n$ where $\mathcal{C}_n$ contains one instance of $\mathcal{T}_T$ in the $q_{p'}$ state. Such computation can be extended in such a way that one instance of $\mathcal{T}_{p'}$ reaches the *on* state and the $\mathcal{T}_T$ component enters in the state q. Let $\mathcal{C}$ be the reached configuration. As the $\mathcal{T}_T$ component is in the state q, we have that $\mathcal{C} \in [\![\vec{m}]\!]$ for some marking $\vec{m}$. Moreover, $m(p') > 0$ because an instance of $\mathcal{T}_{p'}$ is in the *on* state. Fact 2 and Proposition 5 guarantees the reachability in the net RN' of a marking $\vec{m'}$ such that $\vec{m} \leq \vec{m'}$, thus guaranteeing $m'(p') > 0$. □

4.4. Achievability is polynomial in Aeolus$^-$

The achievability problem becomes polynomial in case no capacity constraints are specified on require and provide port, and no conflicts are allowed (i.e., the value 0 on require ports is forbidden). We prove this by presenting a decision algorithm for the achievability problem in Aeolus$^-$.

The underlying idea is to perform an abstract forward exploration of all reachable configurations. Since conflicts cannot be specified the addition to a configuration of new components cannot forbid the execution of previously possible actions. Moreover, since in Aeolus$^-$ provide ports have capacity ∞ and require ports have numerical constraint 1, the correctness of a configuration can be checked simply by verifying that the set of active require ports is a subset of the set of active provide ports.

In the light of the second observation, and knowing that the sets of active require and provide ports are functions of the internal state of the components, we abstractly represent configurations simply as sets of pairs $\langle \mathcal{T}, q \rangle$ indicating the type and the state of the components in the configuration. This way, symbolic configurations abstract away from the exact number of instances of each kind of component, and from their current bindings.

We consider symbolic runs representing the evolutions of abstract configurations. Thanks to the first observation we can restrict ourselves to consider only evolutions where the set of available pairs $\langle \mathcal{T}, q \rangle$ does not decrease. Namely, we perform a symbolic forward exploration starting from an abstract configuration containing all the pairs $\langle \mathcal{T}', \mathcal{T}'.\texttt{init} \rangle$ representing components in their initial state. Then we extend the abstract configuration by adding step-by-step new pairs $\langle \mathcal{T}', q' \rangle$.

Algorithm 1 checks achievability by relying on two auxiliary data structures: *absConf* is the set of pairs $\langle \mathcal{T}', q' \rangle$ indicating the type and state of the components in the current abstract configuration, and *provPort* is the set of provide ports active in such a configuration. The algorithm incrementally extends *absConf* until it is no longer possible to add new pairs. Termination of the algorithm is guaranteed because there are only finitely many type-state pairs in a universe of component types.

Algorithm 1 Verifying achievability in Aeolus$^-$

function ACHIEVABILITY(U, $\mathcal{T}$, q)
  $absConf := \{\langle \mathcal{T}', \mathcal{T}'.\mathtt{init}\rangle \mid \mathcal{T}' \in U\}$
  $provPort := \bigcup_{\langle \mathcal{T}', q'\rangle \in absConf} \{dom(\mathcal{T}'.\mathbf{P}(q'))\}$
  repeat
    $new := \{\langle \mathcal{T}', q'\rangle \mid \langle \mathcal{T}', q''\rangle \in absConf, (q'', q') \in \mathcal{T}'.\mathtt{trans}\} \setminus absConf$
    $newPort := \bigcup_{\langle \mathcal{T}', q'\rangle \in new} \{dom(\mathcal{T}'.\mathbf{P}(q'))\}$
    while $\exists \langle \mathcal{T}', q'\rangle \in new \ . \ dom(\mathcal{T}'.\mathbf{R}(q')) \not\subseteq provPort \cup newPort$ **do**
      $new := new \setminus \{\langle \mathcal{T}', q'\rangle\}$
      $newPort := \bigcup_{\langle \mathcal{T}', q'\rangle \in new} \{dom(\mathcal{T}'.\mathbf{P}(q'))\}$
    end while
    $absConf := absConf \cup new$
    $provPort := provPort \cup newPort$
  until $new = \emptyset$
  if $\langle \mathcal{T}, q\rangle \in absConf$ **then return true**
  else return false
  end if
end function

At each iteration, the potential new pairs are initially computed by checking the automata transitions, and then they are stored in the set *new*. Not all those states could be actually reached as one needs to check whether their require ports are included in the available provide ports *provPort* or in the ports activated by the new states. This is done by a one-by-one elimination of pairs $\langle \mathcal{T}', q'\rangle$ from *new* when their requirements are unsatisfiable. During elimination, we use *newPort* to keep track of the provide ports which are activated by the component states currently in *new*.

When the final set *absConf* is computed, achievability for the component type $\mathcal{T}$ and state q can be simply checked by verifying whether $\langle \mathcal{T}, q\rangle$ is in *absConf*.

We are now ready to prove our polynomiality result for the Aeolus$^-$ model.

Theorem 4. *Let U be a set of component types of the Aeolus$^-$ model. Given the component type $\mathcal{T}$ and the state q, the achievability problem for U, $\mathcal{T}$, and q can be checked in polynomial time (with respect to the size of the descriptions of the components in U).*

Proof. We first prove completeness and soundness of the Algorithm 1, i.e., if a pair $\langle \mathcal{T}, q\rangle$ is achievable then it will be included in *absConf* at the end of the algorithm, and if $\langle \mathcal{T}, q\rangle$ is added to *absConf* then there exists a deployment run to deploy one instance of $\mathcal{T}$ in the q state.

Completeness is proved as follows. The symbolic representation of the initial configuration $\langle U, \emptyset, \emptyset, \emptyset\rangle$ is included in the initial set *absConf*. Consider now a reconfiguration $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$. If the symbolic representation of $\mathcal{C}$ is included in *absConf* at the beginning of an iteration of the **repeat**, then the symbolic rep-

resentation of $\mathcal{C}'$ will be surely included in *absConf* at the end of such iteration. This because the newly reached states in $\mathcal{C}'$ will be also in the *new* set at the end of the **while**. Therefore, if there exists a deployment run able to achieve a component of type $\mathcal{T}$ in the state q, then the algorithm will eventually include the pair $\langle \mathcal{T}, q \rangle$ in *absConf*.

The soundness is proved as follows. Let $absConf_i$ be the set *absConf* at the end of the i-th iteration of the **repeat**, and let $absConf_0$ be the set containing only the pairs $\langle \mathcal{T}', \mathcal{T}'.\mathtt{init} \rangle$. By induction on i, we prove that there exists a deployment run that includes at least one instance for every type-state pair in $absConf_i$. For the base case $i = 0$, this trivially holds (it is sufficient to consider a deployment run that creates at least one instance for each component type). In the induction case we consider $absConf_i$, and by induction hypothesis we assume the existence of a deployment run $\mathcal{C}_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} \mathcal{C}$ that generates at least one instance for the type-state pairs in $absConf_i$. We show the existence of a deployment run for the pairs in $absConf_{i+1}$. Consider the new pairs $\langle \mathcal{T}', q' \rangle$ added in $absConf_{i+1}$ and let $\mathcal{P}$ be the multiset of pairs necessary to generate such new pairs, i.e.,

$$\mathcal{P} = \{\!\{ \langle \mathcal{T}', q'' \rangle \mid (q'', q') \text{ is the transition used to add } \langle \mathcal{T}', q' \rangle \text{ to } absConf \}\!\}$$

Consider now a deployment run obtained by repeating the actions $\alpha_1, \cdots, \alpha_n$ of the deployment run $\mathcal{C}_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} \mathcal{C}$ until the reached configuration contains at least one instance for the pairs in $absConf_i$ plus one additional instance for each occurrence of the pairs in the multiset $\mathcal{P}$. This deployment run can be extended with state changes of these additional instances to reach the new $\langle \mathcal{T}', q' \rangle$ pairs.

We now show that the complexity of the algorithm is polynomial w.r.t. the size of the description of the universe of component types U. Polynomiality is guaranteed by the fact that both the **repeat** and the **while** cycles perform a number of iterations smaller than the number of different pairs $\langle \mathcal{T}', q' \rangle$ in the universe U. □

5. Related work

To the best of our knowledge Aeolus is the first model that is designed on purpose to formally addresses the specific needs of software component deployment in the cloud. It was first introduced in [1] and further developed within the ANR project Aeolus "Mastering the Complexity of the Cloud" [19, 20]. Differently from the definition of the language presented here, in [1] an additional kind of requirements—called *weak requirements*—was present. Whereas the requirements presented in this paper (formerly known as *strong requirements*) need to be enforced every step, weak requirements are verified only at the end of a deployment run and thus need to be satisfied only in the final configuration. We decided to drop the notion of weak requirements for two reasons: firstly, we noticed that they were not very used in practice; secondly, their behaviour can

easily be simulated with strong requirements, so there was no real gain in terms of model expressivity.[3]

This paper improves the complexity result about achievability in the Aeolus core model, which was first proven decidable in [2]. In that paper, reconfigurability (the generalization of achievability with any initial configuration, also the non empty one) was proved to be ExpSpace-hard by reduction from the coverability problem in standard Petri nets; here we have considered reset Petri nets thus proving that the problem is furthermore Ackermann-hard.

5.1. Formal models

We now compare the Aeolus approach to related formal models that have been proposed in slightly different contexts.

Automata have been adopted long ago in the context of component-oriented development frameworks. One of the most influential model are *interface automata* [21], where automata are used to represent the component behaviour in terms of input, output, and internal actions. Interface automata support automatic compatibility check and refinement verification: a component refines another if its interface has weaker input assumptions and stronger output guarantees. Differently from that approach, we are not interested in component compatibility or refinement, and we do not require complementary behaviour of components. We simply check in the current configuration whether all required functionalities are provided by currently deployed components. The automata in Aeolus do not represent the internal behaviour of components, but the effect on the component of an external deployment or reconfiguration actions.

Aeolus reconfiguration actions show interesting similarities with transitions in Petri nets [22], a very popular model already presented in the previous sections and born from the attempt to extend automata with concurrency. At first sight one might encode our model in Petri nets, representing Aeolus component states as places, each deployed component as a token in the corresponding place, and reconfiguration actions as transitions that cancel and produce tokens. Achievability in Aeolus would then correspond to *coverability* in Petri nets. But there are several important differences. Multiple state change actions can atomically change the state of an unbounded number of components, while in Petri net each transition consumes a predefined number of tokens. More importantly, we have proved that achievability can be solved in polynomial time for the Aeolus$^-$ fragment and that it is undecidable for the Aeolus model, while in Petri nets coverability is an ExpSpace problem [23].

[3] In order to impose requirements only on the final configuration one can duplicate every state q of the components by adding a new corresponding state q'. The state q' will then provide and require the same ports of q, and can be reached from q via a transition. Intuitively, q' states should be the ones that the components reach at the end of the plan. A weak requirement of state q in this case is modelled as a requirement of state q'. To impose that at the end of the deployment run only states q' are present it is enough to add to every q state a provide port that is in conflict with the target state of the desired component.

Several process calculi extend/modify the π-calculus [24] in order to deal with software components. The Piccola calculus [25] extends the asynchronous π-calculus [24] with *forms*, first-class extensible namespaces, useful to model component interfaces and bindings. Calculi like KELL [26] and HOMER [27] extends a core π-calculus with hierarchical locations, local actions, higher-order communication, programmable membranes, and dynamic binding. More recently, MECo [28] has extended this approach by proposing also explicit component interfaces and channels to realize tunnelling effects traversing the hierarchical location boundaries. On the one hand, all these proposals differ from Aeolus in that they focus on the modelling of component interactions and communication, while we focus on their interdependencies during system deployment and reconfiguration. On the other hand, we plan to take inspiration from these calculi in order to extend our model with boundaries and administrative domains.

We have already briefly discussed in the introduction the Fractal component [7], and the related cloud middleware FraSCAti [8]. In terms of the underlying formal model, we observe that Fractal does provide an object-oriented API to manage the life-cycle of components which, in spirit, is close to what Aeolus aims to do with component automata. However, the OO API approach is more limited when it comes to the ability to reason on component activation: in Aeolus we can, within limits due to the problem complexity, reason on and automate component activation; in Fractal an external reasoner will have to stop at API invocation, without knowledge of what a specific method implementation will do. Also, in Aeolus component states are not limited to active/inactive, i.e., each component type can define its own life-cycle in detail.

A declarative approach similar to Aeolus for modelling individual components of a system, together with their possible configuration states, is also introduced in the position paper [29]. However, the lack of a formal semantics for the approach makes impossible to analyse the complexity of the deployment problem in their setting. Moreover, as observed by the authors, their approach allows administrators to write erroneous models presenting deadlocks or livelocks that are difficult to detect and forbid the reaching of the desired target configuration.

5.2. Tools

The complementary tools Zephyrus and Metis, available at [30], are directly related to the Aeolus model. Zephyrus [31] tackles the problem of computing a valid system configuration (according to the Aeolus model), starting from an existing configuration, a universe of available component types, and a formal specification that captures user desiderata for the target system. Furthermore, Zephyrus also takes into account limited machine resources, such as CPU, memory, bandwidth, etc. The computation is done via translation to a set of integer constraints, plus a dedicated algorithm to compute bindings, and the approach makes it is possible to add an objective function that can be minimized or maximized to optimize the resulting configuration.

Metis [32, 33] tackles instead the problem of quickly computing a plan that migrates a valid Aeolus configuration into a different one (possibly synthesized by Zephyrus). The authors consider a model similar to Aeolus$^-$ without the possibility of using multi-state changes and propose an algorithm to compute a deployment run to reach a given target state in a specific component. The algorithm has been proven to be sound, complete, and polynomial in time w.r.t. the size of the encoding of the components in the universe.

In the same area of Zephyrus and Metis we find various tools, coming from both industry and academia.

In [9] the ConfSolve tool is presented by showing how constraint solving techniques can be used to propose an optimal allocation of virtual machines to servers, and of applications to virtual machines. An object-oriented declarative language is used to describe the entities (e.g., machines and services), the constraints, and the optimisation criteria. A final configuration is then computed by translating the declarative specification into a constraint optimisation problem which is then solved by using a constraint solver. Similarly to Zephyrus, but differently from Metis, ConfSolve does not consider the problem of synthesising a low-level plan to reach the final configuration.

Off-the shelf planning solvers are exploited in [34, 35] to generate automatically the actions to reconfigure a system. To use these tools, however, all the deployment actions with theirs preconditions and effects need to be properly specified. This hinders the usability of these kinds of tools. Indeed, to use them, system administrators are forced to translate their requirements and specifications into a formalism similar to the one used by the Planning Domain Definition Language (i.e., the *de facto* standard language for classical planners). Aeolus does not relieve administrators from the need of expressing the dynamic behavior of components, but allows to do so more succinctly, and in a formalism that is independent from low-level planning tools.

Two recent efforts, Juju and Engage, are similar to Zephyrus, but they both avoid the problem of dealing with conflicts. In Juju [6], each service is deployed on a single machine (or, more recently, in a virtual container). This avoids the issue of component incompatibilities, but does so at the price of potentially wasting resources. Engage [36] relies on a solver to plan deployments, but offers no support for conflicts in the specification language: one can only indicate that a service can be realized by exactly one out of a list of components. Furthermore, neither Juju nor Engage—or any other tool that we are aware of—allows to declare capacity or replication constraints, which are essential non functional constraints for any non-trivial, scalable application.

6. Conclusions

The Aeolus formalism is a component model designed specifically to capture most common deployment scenarii for distributed software applications in the cloud. It allows to study formally the operations that are needed to deploy complex applications on modern computing infrastructure.

In this work, we have provided precise complexity results for several variants of the Aeolus component model. We have shown that it is possible to generate a deployment plan in polynomial time for the fragment *Aeolus*$^-$ that corresponds to the limited industrial tools currently in use.

Adding support for defining conflicts among components corresponds to using the fragment *Aeolus core*. We have shown that, within such fragment, it is still possible to automatically generate a plan, at the price of a very high worst case complexity, as the problem becomes Ackermann-hard.

As for the generation of deployment plans in the full generality of the *Aeolus* model, we have presented an undecidability result, which provides a clear limit to what automated tools can do.

We plan to investigate realistic restrictions on the *Aeolus* model for which efficient reconfigurability algorithms could still be devised. For instance, one can consider imposing an upper limit on the number of resources that can be allocated during a deployment run, or investigate the impact of restricting the shape or size of the internal state machine of the components.

It will also be interesting to extend the *Aeolus* model to take into account nested components or administrative domains and explore the impact of such extensions on the complexity of the generation of deployment plans.

From a practical point of view, we started to create a repository of software components with their *Aeolus* metadata. In particular, our industrial partner Mandriva is currently enriching the description of the packages used in their industrial products to automatically deploy them using Aeolus technologies.

References

[1] R. Di Cosmo, S. Zacchiroli, G. Zavattaro, Towards a Formal Component Model for the Cloud, in: SEFM 2012, Vol. 7504 of LNCS, Springer, 2012, pp. 156–171.

[2] R. Di Cosmo, J. Mauro, S. Zacchiroli, G. Zavattaro, Component Reconfiguration in the Presence of Conflicts, in: ICALP 2013: 40th International Colloquium on Automata, Languages and Programming, Vol. 7966 of LNCS, Springer-Verlag, 2013, pp. 187–198.

[3] L. Kanies, Puppet: Next-generation configuration management, `;login:` the USENIX magazine 31 (1) (2006) 19–25.

[4] Opscode, Chef, http://www.opscode.com/chef/.

[5] Cloud Foundry, http://cloudfoundry.org/.

[6] Juju, devops distilled, https://juju.ubuntu.com/.

[7] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, J.-B. Stefani, The FRACTAL component model and its support in Java, Softw., Pract. Exper. 36 (11-12) (2006) 1257–1284.

[8] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, J.-B. Stefani, Reconfigurable SCA applications with the FraSCAti platform, in: IEEE SCC, IEEE, 2009, pp. 268–275.

[9] J. A. Hewson, P. Anderson, A. D. Gordon, A Declarative Approach to Automated Configuration, in: LISA '12: Large Installation System Administration Conference, 2012, pp. 51–66.

[10] P. Abate, R. Di Cosmo, R. Treinen, S. Zacchiroli, MPM: a modular package manager, in: CBSE'11: 14th symposium on component based software eng., ACM, 2011, pp. 179–188.

[11] K. Schmid, A. Rummler, Cloud-based software product lines, in: SPLC, ACM, 2012, pp. 164–170.

[12] R. Di Cosmo, P. Trezentos, S. Zacchiroli, Package upgrades in FOSS distributions: Details and challenges, in: HotSWup'08, 2008.

[13] K.-K. Lau, Z. Wang, Software component models, IEEE Trans. Software Eng. 33 (10) (2007) 709–724.

[14] M. Minsky, Computation: finite and infinite machines, Prentice Hall, 1967.

[15] P. A. Abdulla, K. Cerans, B. Jonsson, Y.-K. Tsay, General decidability theorems for infinite-state systems, in: LICS, IEEE, 1996, pp. 313–321.

[16] A. Finkel, P. Schnoebelen, Well-structured transition systems everywhere!, Theoretical Computer Science 256 (2001) 63–92.

[17] L. E. Dickson, Finiteness of the Odd Perfect and Primitive Abundant Numbers with n Distinct Prime Factors, American Journal of Mathematics 35 (4) (1913) 413–422.

[18] P. Schnoebelen, Revisiting Ackermann-Hardness for Lossy Counter Machines and Reset Petri Nets, in: MFCS, Vol. 6281 of Lecture Notes in Computer Science, Springer, 2010, pp. 616–628.

[19] M. Catan, R. D. Cosmo, A. Eiche, T. A. Lascu, M. Lienhardt, J. Mauro, R. Treinen, S. Zacchiroli, G. Zavattaro, J. Zwolakowski, Aeolus: Mastering the Complexity of Cloud Application Deployment, in: ESOCC, Vol. 8135 of Lecture Notes in Computer Science, Springer, 2013.

[20] Aeolus: Mastering the Complexity of the Cloud, http://www.aeolus-project.org/.

[21] L. de Alfaro, T. A. Henzinger, Interface automata, in: ESEC / SIGSOFT FSE, 2001.

[22] C. A. Petri, Kommunikation mit Automaten, PhD thesis, Institut für Instrumentelle Mathematik, Bonn, Germany, 1962.

[23] C. Rackoff, The covering and boundedness problems for vector addition systems, Theoret. Comp. Sci. 6 (1978) 223–231.

[24] R. Milner, J. Parrow, D. Walker, A Calculus of Mobile Processes, I/II, Inf. Comput. 100 (1) (1992) 1–77.

[25] F. Achermann, O. Nierstrasz, A calculus for reasoning about software composition, Theor. Comput. Sci. 331 (2-3) (2005) 367–396.

[26] A. Schmitt, J.-B. Stefani, The Kell Calculus: A Family of Higher-Order Distributed Process Calculi, in: Global Computing, Vol. 3267 of LNCS, Springer, 2004, pp. 146–178.

[27] M. Bundgaard, T. T. Hildebrandt, J. C. Godskesen, A CPS encoding of name-passing in Higher-order mobile embedded resources, Theor. Comput. Sci. 356 (3) (2006) 422–439.

[28] F. Montesi, D. Sangiorgi, A Model of Evolvable Components, in: TGC, Vol. 6084 of Lecture Notes in Computer Science, Springer, 2010, pp. 153–171.

[29] P. Goldsack, P. Murray, A. Farrell, P. Toft, SmartFrog and Data Centre Automation, Tech. rep., Microsoft Research (2008).

[30] Aeolus Tools, https://github.com/aeolus-project/.

[31] R. Di Cosmo, M. Lienhardt, R. Treinen, S. Zacchiroli, J. Zwolakowski, Optimal Provisioning in the Cloud, Tech. rep., Aeolus project, http://hal.archives-ouvertes.fr/hal-00831455 (June 2013).

[32] T. A. Lascu, J. Mauro, G. Zavattaro, Automatic Component Deployment in the Presence of Circular Dependencies, in: The 10th International Symposium on Formal Aspects of Component Software, FACS 2013.

[33] T. A. Lascu, J. Mauro, G. Zavattaro, A Planning Tool Supporting the Deployment of Cloud Applications, in: ICTAI, IEEE, 2013, pp. 213–220.

[34] H. Herry, P. Anderson, G. Wickler, Automated Planning for Configuration Changes, in: LISA, USENIX Association, 2011.

[35] H. Herry, P. Anderson, Planning with Global Constraints for Computing Infrastructure Reconfiguration, in: CP4PS-12 - The AAAI-12 Workshop on Problem Solving using Classical Planners, 2012.

[36] J. Fischer, R. Majumdar, S. Esmaeilsabzali, Engage: a deployment management system, in: PLDI'12: Programming Language Design and Implementation, ACM, 2012, pp. 263–274.

CHAPTER 11

Automated Synthesis and Deployment of Cloud Applications

This chapter contains the full text of the article "Automated Synthesis and Deployment of Cloud Applications" *[50].*

Automated Synthesis and Deployment of Cloud Applications *

Roberto Di Cosmo
roberto@dicosmo.org

Michael Lienhardt
michael.lienhardt@inria.fr

Ralf Treinen
treinen@pps.univ-paris-diderot.fr

Stefano Zacchiroli
zack@pps.univ-paris-diderot.fr

Jakub Zwolakowski
jakub.zwolakowski@inria.fr

Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126, CNRS, F-75205 Paris, France

Antoine Eiche
Mandriva, FR
aeiche@mandriva.com

Alexis Agahi
Kyriba Corporation, USA
alexis.agahi@kyriba.com

ABSTRACT

Complex networked applications are assembled by connecting software components distributed across multiple machines. Building and deploying such systems is a challenging problem which requires a significant amount of expertise: the system architect must ensure that all component dependencies are satisfied, avoid conflicting components, and add the right amount of component replicas to account for quality of service and fault-tolerance. In a cloud environment, one also needs to minimize the virtual resources provisioned upfront, to reduce the cost of operation. Once the full architecture is designed, it is necessary to correctly orchestrate the deployment phase, to ensure all components are started and connected in the right order.

We present a toolchain that automates the assembly and deployment of such complex distributed applications. Given as input a high-level specification of the desired system, the set of available components together with their requirements, and the maximal amount of virtual resources to be committed, it synthesizes the full architecture of the system, placing components in an optimal manner using the minimal number of available machines, and automatically deploys the complete system in a cloud environment.

*This work was supported by the French ANR project ANR-2010-SEGI-013-01 Aeolus and partially performed at IRILL, center for Free Software Research and Innovation in Paris, France, `http://www.irill.org`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE'14, September 15-19, 2014, Vasteras, Sweden.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2642980.

1. INTRODUCTION

In contrast to classic, monolithic software that runs locally on a single machine, large distributed systems are built from many *running services* executing on (possibly heterogeneous) *virtual machines* (or *locations*) and collaborating to provide the expected functionality to final users.

The system *architect* must choose which services to use and how to configure them, knowing that services may depend on, and/or be in conflict with, each other; consider fault tolerance and quality of service issues, and provide enough instances of each service; design the physical architecture on which to run the system, trying to keep its *cost* reasonable with nonetheless enough locations with enough *resources* (e.g. RAM, disk space, bandwidth) to allow the installation and the good execution of the services they host; choose which implementation of each service to install on which location, knowing that implementations (usually in the form of *packages*) have dependencies and conflicts too. Once all this planning is done, the *deployment* phase must provision the required virtual machines, install the right packages on each of them, and finally start and interconnect services in the right order.

This is a daunting task, not unlike building a puzzle—each running service, package, and machine being a piece—where one only knows the overall expected functionality.

To reduce the complexity of this process, many industrial initiatives develop tools [32, 5] that allow to select, configure, and push on the Cloud some well defined services. However, these tools are only useful once the puzzle is solved, i.e. when the right services and packages have been selected, the locations on which they must be deployed have been chosen, and the way of configuring them in a manner that satisfies all the requirements has been found. Solving this puzzle currently requires a significant amount of human expertise, so that in practice large software stacks are often managed using custom scripts and manual techniques, which are error-prone and fragile [23].

In this article, we present a toolchain developed in the framework of the Aeolus project [6] that provides a generic, automatic and sound alternative to these scripts and techniques. The toolchain is composed of two tools:

Zephyrus automatically generates an abstract representation (or *configuration*) of the target system according to a concise specification of the expected functionalities. It takes into account the set of available services, which can serve as building blocks, with their requirements, replication policies, and resource consumption characteristics; information concerning the implementation of the services (e.g. that the Apache service is provided by the www-servers/apache package on Gentoo Linux, by the apache2 package on Debian, etc.); and the maximum amount of (virtual) machines available, together with their characteristics. It is also able to minimize the amount of needed resources.

Armonic takes the full system configuration produced by Zephyrus and deploys it, by provisioning the required virtual machines on a cloud computing platform (such as OpenStack), installing the needed packages on each machine; it configures the different services to establish the required connections; and finally starts the services in the right order, relying on precise metadata that describe the internal state machines and runtime dependencies of each service.

This toolchain decouples system design from system deployment. It builds upon the sound formal foundations of the Aeolus component model [9], which describes each available service as a *component type*, using *ports* tagged with an *arity* to encode requirements, provides, conflicts and replication policy, as well as an internal state machine to capture component life-cycles.

Zephyrus uses a stateless version of the Aeolus model, extended to take into account locations, repositories, packages, and resources, as detailed in [7]; packages and repositories are encoded following the now standard approach originated from [21]. The specifications accepted by Zephyrus are given in a rigorous syntax whose semantics defines when a configuration satisfies a specification. Based on this formalization, Zephyrus can be proven correct and complete: it will always find a configuration that is optimal w.r.t. the chosen criterion if one exists. Furthermore, the generated configuration is guaranteed to provide the expected functionalities, and satisfies the constraints defined by the replication policies, as well as the dependencies and conflicts between services.

Armonic then takes over and uses the information about the internal state machine of the components to determine a correct service activation sequence, under the assumption that the dependency relation among services is acyclic, which is usually the case.

Paper structure. Section 2 presents the toolchain through a realistic running example. Section 3 discusses the internals of the various tools and presents the theoretical results establishing soundness, completeness and complexity of the architecture synthesis phase. Section 4 reports on experimentation with the toolchain, as well as its adoption in an industrial context at Kyriba Corporation. Before concluding, Section 5 discusses related work.

2. WALKTHROUGH

In this section we describe Zephyrus and Armonic by showing them at work on an example that is simple enough to be fully presented, and yet realistic as it corresponds to a common use case of application deployment in the cloud.

Zephyrus takes several inputs:

1. a description of all the existing components and their constraints, which come in various formats due to their different origins (e.g. package database, architectural choices, machine physical resources, etc.); this is called a *universe*.
2. a description of the *current system configuration* (existing machines, which services are currently deployed where, etc.)
3. a high level *specification* of the desired system. As part of the specification, architects can include objective functions that they would like to optimize for, such as the desire of minimizing the number of virtual machines that will be used for the deployment (and hence the system cost).

2.1 Use case: deploying a WordPress farm

The task we want to perform is deploying the popular Wordpress blog platform on a private OpenStack cloud. In addition to being realistic, this use case is often used as a "benchmark" to showcase the characteristics of cloud provisioning platforms. Wordpress is written in PHP and as such is executed within Web server software like Apache or nginx. Additionally, Wordpress needs a connection to a MySQL instance, in order to store user data. Simple Wordpress deployments can therefore be obtained on a single machine where both Wordpress and MySQL get installed.

"Serious" Wordpress deployments, however—that sustain high load and are fault tolerant—are more complex and rely on some form of load balancing. One possibility is to balance load at the DNS level using servers like Bind: multiple DNS requests to resolve the website name will result in different IPs from a given pool of machines, on each of which a separate Wordpress instance is running. Alternatively one can use as website entry point an HTTP reverse proxy capable of load balancing (and caching, for added benefit) such as Varnish. Either way, Wordpress instances will need to be configured to contact the same MySQL database, to avoid delivering inconsistent results to users. Also, having redundancy and balancing at the front-end level, one usually expects to have them also at the DBMS level. One way to achieve that is to use a MySQL *cluster*, and configure the Wordpress instances with multiple entry points to it.

Constraints. Several design constraints should be taken into account when designing such a system. Some constraints come from package providers and cannot be easily changed. For instance, Wordpress, Varnish, etc. usually come from software distribution packages and have their own dependencies and conflicts which must be respected on each machine when installing the software.

On the other hand, "house" requirements are defined by system architects to capture some ad-hoc policy. For this use case, we assume given the following requirements:

- at least 3 replicas of Wordpress behind Varnish or, alternatively, at least 7 replicas with DNS-based load balancing (since DNS-based load balancing is not capable of caching, the expected load on individual Wordpress instances is higher);
- at least 2 different entry points to the MySQL cluster;
- each MySQL instance shouldn't serve the needs of more than 3 Wordpress instances;

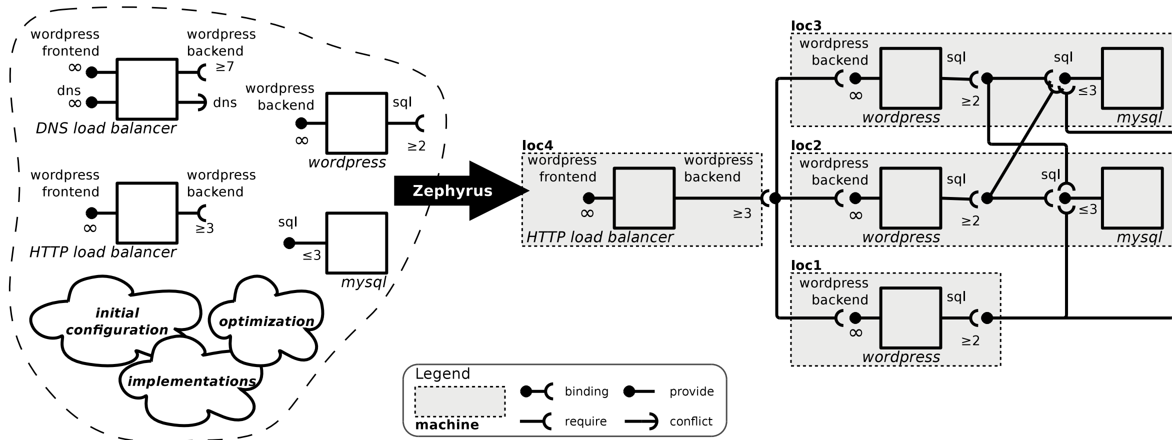


Figure 1: `Zephyrus` used to design a scalable, fault-tolerant Wordpress deployment

- no more than 1 DNS server deployed in the administrative domain;
- different Wordpress (and MySQL) instances are deployed at different locations.[1]

Similar constraints might exist on machine resources, e.g. we expect Varnish to consume 2Gb of RAM and we don't want to deploy it to a smaller machine, especially if in combination with other RAM-consuming services. Note that "house" requirements are not intrinsically related to the software components we are using, but are rather an encoding of explicit architectural choices.

2.2 Architecture synthesis

Figure 1 shows the application of our toolchain to the design of a complex Wordpress deployment like the one we have discussed.

On the left of the black arrow is a schematic representation of `Zephyrus`' input, on the right its output. Available services are depicted in the figure using a graphical syntax inspired by the Aeolus model [9], each one with its own requirements, conflicts, and (house) replication policy. Component requirements are exposed as required ports that should be connected, via bindings, to matching provided ports offered by other service instances, respecting port replication constraints: an upper bound (or ∞) on the amount of incoming bindings for provided ports; a lower bound on the amount of outgoing bindings to *different* service instances for required ports. For example, the fact that the HTTP load balancer requires 3 Wordpress replicas is indicated by the ≥ 3 annotation on its `wordpress backend` required port, and the fact that the DNS load balancer is incompatible with other DNS services is indicated by the `dns` conflict port.

Note that our ports result in a very flexible notion of dependency, with *choice*: any requirement can be satisfied by *any* component providing the right port. For instance, if we require the port `wordpress frontend`, we allow `Zephyrus` to choose which of `DNS load balancer` or `HTTP load balancer` is the best to use. To our knowledge, `Zephyrus` is the only tool to manage such flexibility in dependencies.

[1]It is technically possible to co-locate multiple, say, MySQL instances on the same machine, but it would be pointless to do so when we are seeking fault tolerance and load balancing.

Services and implementations. `Zephyrus` takes as input a description of the available service types, and an *implementation* relation that maps each *service* to the set of packages implementing it.[2] These two parts of the universe are given as input to `Zephyrus` as a JSON file that in our running example looks like this:

```
{ "component_types": [
  { "name"    : "DNS-load-balancer",
    "provide" : [["@wordpress-frontend"], ["@dns"]],
    "require" : [["@wordpress-backend", 7]],
    "conflict": ["@dns"],
    "consume" : [["ram", 128]] },
  { "name"    : "HTTP-load-balancer",
    "provide" : [["@wordpress-frontend"]],
    "require" : [["@wordpress-backend", 3]],
    "consume" : [["ram", 2048]] },
  { "name"    : "Wordpress",
    "provide" : [["@wordpress-backend"]],
    "require" : [["@sql", 2]],
    "consume" : [["ram", 512]] },
  { "name"    : "MySQL",
    "provide" : [["@sql", 3]],
    "consume" : [["ram", 512]] } ],
  "implementation": [
    [ "DNS-load-balancer", ["bind9"] ],
    [ "HTTP-load-balancer", ["varnish"] ],
    [ "Wordpress", ["wordpress"] ],
    [ "MySQL", ["mysql-server"] ] ] }
```

The `component_types` section describes the available component types with their ports, as well as their non functional requirements like memory or bandwidth. Port names are distinguished from components or packages by a simple syntactic convention: ports start with `@`. The `implementation` section maps services to the software packages that should be installed to realize them on actual machines.

Package repositories. Unlike other tools, `Zephyrus` is fully aware of available package repositories, with their dependencies and conflicts, and uses such information to ensure that package-level conflicts and dependencies are respected on all machines. It is possible to associate different package repos-

[2]In the example we have kept things simple, but `Zephyrus` is capable of handling complex situations where the same service can be implemented by different packages on different machines, according to the locally installed OS.

Table 1 Specification Syntax

S	::=	true $\mid$ $e\ op\ e$	Specification
	$\mid$	S and S $\mid$ S or S	
	$\mid$	S => S $\mid$ not S	
e	::=	$n \mid \#\ell \mid n \times e$	Expression
	$\mid$	$e + e \mid e - e$	
ℓ	::=	$k \mid t \mid p$	Elements
	$\mid$	$(J_\phi)\{J_r : S_l\}$	
S_l	::=	true $\mid$ $e_l\ op\ e_l$	Local Specification
	$\mid$	S_l and S_l $\mid$ S_l or S_l	
	$\mid$	S_l => S_l $\mid$ not S_l	
e_l	::=	$n \mid \#\ell_l \mid n \times e_l$	Local Expression
	$\mid$	$e_l + e_l \mid e_l - e_l$	
ℓ_l	::=	$k \mid t \mid p$	Local Elements
J_ϕ	::=	$_ \mid o\ op\ n; J_\phi$	Resource Constraint
J_r	::=	$_ \mid r \mid r \vee J_r$	Repository Constraint
op	::=	$\leq \mid = \mid \geq$	Operators

itories to different locations, allowing to handle deployment of heterogeneous systems. The size of a repository may be huge (the Debian Squeeze repository contains ≈30'000 packages), so `Zephyrus` uses `coinst` [8] to abstract packages into a set of much smaller equivalence classes, and yet sufficient to capture all package incompatibilities.

Available (virtual) machines. Another essential part of `Zephyrus` input is the description of the initial configuration, i.e. the set of available machines with information on their resources: memory, package repository, existing services and packages. In our example, we start with an initial configuration consisting of 6 bare locations with 2Gb of RAM. Such configuration is fed to `Zephyrus` in JSON format, e.g.:

```
{ "locations" : [
   { "name" : "loc1",
     "repository" : "debian-squeeze",
     "provide_resources" : [["ram", 2048]] },
   { "name" : "loc2",
     "repository" : "debian-squeeze",
     "provide_resources" : [["ram", 2048]] },
   [...] }
```

Target system specification. `Zephyrus` accepts a specification of the desired target system. Specifications are defined according the abstract syntax presented in Table 1.

A specification S is a set of basic constraints $e\ op\ e$, combined using the usual logical connectors. These basic constraints specify how many elements (packages, component types, etc) should be in the generated configuration, using terms of the form $\#\ell$ that correspond to the number of instances of element ℓ in the system. For instance, one might state that we want at least 3 instances of the component type `apache`: "`#apache` $\geq$ `3`", where `#apache` represents the number of `apache` instances in the configuration.

Moreover, it is possible to express constraints on locations. Locations can be specified in our syntax with the term $(J_\phi)\{J_r : S_l\}$ where J_ϕ is the constraint on the resource available on that machine; J_r is the set of repositories that can be installed on that machine ('_' standing for any repository); and S_l is a constraint specifying the contents of the machine (basically, S_l is S without locations). For instance, we can specify that no location with less than 2Gb of RAM and `redhat` installed should have a `MySQL` running: "`#(mem < 2G){redhat: #MySQL` $\geq$ `1} = 0`".

For our running example we need exactly one Wordpress frontend (i.e., exactly one service offering a `wordpress-frontend` port), and that no machine is deployed with more than one instance of either MySQL/Wordpress services on it.

```
(#@wordpress-frontend = 1)
and #(_){_ : #MySQL > 1} = 0
and #(_){_ : #Wordpress > 1} = 0
```

Note that no constraint is imposed on the co-location of *different* services on the same machine.

Optimization criteria. In `Zephyrus`, one may request a solution that is optimal w.r.t. a specific objective function. Currently, `Zephyrus` supports two built-in optimization criteria, namely `compact` and `conservative`, which respectively minimize the number of components and locations used, or their *difference* with respect to the initial configuration.

Running Zephyrus. We are now ready to ask `Zephyrus` to compute the final configuration:

```
$ zephyrus -u univ.json -opt compact \
  -ic conf.json -spec sp.spec \
  -repo debian-squeeze ds.coinst
```

In addition to the obvious parameters (universe, optimization function, configuration, specification), we pass an extra one: the `-repo` option tells `Zephyrus` that all the information about the packages contained in the repository named `debian-squeeze` is available in the file `ds.coinst`.

The actual output of `Zephyrus` contains a complete description of the system to be deployed; it is too long to be listed here in full, so we only highlight some excerpts of it. The format is the same as for configurations, and starts with the description of the locations:

```
{ "locations": [
   { "name": "loc1",
     "provide_resources": [ [ "ram", 2048 ] ],
     "repository": "debian-squeeze",
     "packages_installed": [ "wordpress" ] },
   { "name": "loc2",
     "provide_resources": [ [ "ram", 2048 ] ],
     "repository": "debian-squeeze",
     "packages_installed": ["mysql-server",
       "wordpress" ] }                          [...]
```

We see that each location is associated to a list of packages that should be installed there. Only the root packages are listed, and `Zephyrus` has already checked that they can be co-installed, satisfying dependencies and conflicts.

The second part of the output is the list of service instances present in the system, mapped to their locations:

```
 "components": [
   { "name": "Wordpress-1",
     "type": "Wordpress",
     "location": "loc1" },
   { "name": "Wordpress-2",
     "type": "Wordpress",
     "location": "loc2" },
   { "name": "MySQL-1", "type": "MySQL",
     "location": "loc2" },               [...]
```

Finally, the third part of the output lists the bindings that connect (ports of) service instances together:

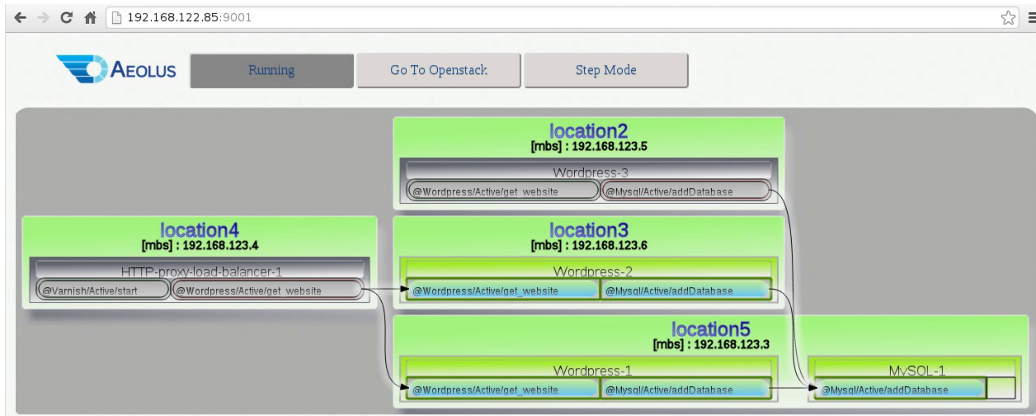


Figure 2: Screenshot of `Armonic` web interface.

```
"bindings": [
  { "port": "@wordpress-backend",
    "requirer": "HTTP-load-balancer-1",
    "provider": "Wordpress-1" },
  { "port": "@wordpress-backend",
    "requirer": "HTTP-load-balancer-1",
    "provider": "Wordpress-2" },
  { "port": "@sql",
    "requirer": "Wordpress-1",
    "provider": "MySQL-1" }              [...]
```

The configuration corresponding to `Zephyrus` output is depicted on the right of Figure 1, where shaded boxes denote locations; we omit installed packages for the sake of readability. All choices there—load balancer, mapping between services and machines, bindings, etc.—have been made by `Zephyrus`. Note how services have been co-located where possible, minimizing the number of used machines: only 4 out of the 6 available machines have been used: the proposed solution is optimal w.r.t. the desired metric.

2.3 Configuration deployment

Once we have the configuration generated by `Zephyrus` as output, we are left with the task of turning it into a running system. While `Zephyrus` output is agnostic as to the final deployment tool, we have developed our own tool—called `Armonic`—to work closely in conjunction with `Zephyrus`.

Starting from the configuration generated by `Zephyrus`, `Armonic` first provisions the needed virtual machines (VMs) on a target private OpenStack cloud, creating new instances as needed. The resource information (CPU, storage, memory) contained in `Zephyrus` output are used by `Armonic` to choose appropriately-sized VMs (AKA "flavors").

After provisioning, `Armonic` takes care of *configuring VMs* using an agent-orchestrator architecture: each VM comes with an agent which receives configuration instructions from a central orchestrator. During VM configuration, `Armonic` installs on each VMs the packages dictated by `Zephyrus`; tunes service configuration files to implement bindings (e.g. to "connect" a Wordpress to a MySQL, `Armonic` will patch a Wordpress configuration file to point to a given VM IP address and port); and starts services in the right order as it goes. In our example, a Wordpress instance is deployed at location `loc1`. However, a MySQL database must be available *before* the deployment of Wordpress in order to properly start the service. To address this, `Armonic` devises a *deployment plan*, using bindings to determine a suitable deployment ordering, and follows it during component deployment. In the example `Armonic` will deploy Mysql right away (as it has no further component dependencies), then Wordpress, and finally the load balancer.

The `Armonic` orchestrator is equipped with a web interface, shown in Figure 2 at work on the deployment of a simplified version (using a single MySQL database, instead of a cluster) of the configuration of Figure 1. To deploy an application, users can simply feed `Zephyrus` output into the `Armonic` web interface and then monitor live the state of deployment. In Figure 2 the deployment is almost finished: all components are deployed and connected, except the last Wordpress instance and the load balancer which are shown grayed-out, as they are only partially deployed. The deployment of this use case takes about 7 minutes, including building and booting virtual machines, package installation, and service configuration.

3. TOOLCHAIN INTERNALS

3.1 Minimizing input

Figure 3 presents a simple schema of the architecture of `Zephyrus`, which is basically structured into five blocks. The input phase of `Zephyrus` collects all the data provided by the user.

For each location, we take into account not only the available services, but also all the possible ways to deploy them (i.e. packages that must be installed to realize them, together with their dependencies): this can amount to handle tens of thousands of packages for each location, and a naïve approach would simply be unfeasible. We believe this is one of the fundamental reasons why competing tools do not represent package relationships explicitly or completely, with the consequence of potentially producing configurations which are not deployable due to package incompatibilities unknown to the tool. We compare this aspect of `Zephyrus` with alternative approaches in Section 5.

To render the problem tractable, `Zephyrus` performs several simplification passes on the input data that greatly reduce its size: the universe is trimmed by removing all services that are not in the transitive closure of the services present in the initial configuration or the request; package

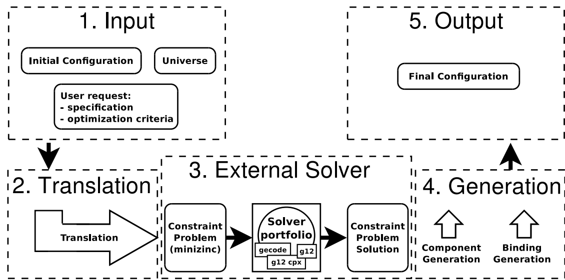


Figure 3: Overall architecture of Zephyrus

repositories are pruned by keeping only packages that implement services which were not removed by the previous simplification phase; lower and upper bounds on the needed resources and components are computed, and only the minimum estimated number of available locations is kept. All these operations are safe, as we can prove that they do not exclude any correct solution.

A second important simplification is achieved by using a slightly modified version of the `coinst` tool [8], which reduces by several orders of magnitude the data present in software package repositories, like those offered by the Debian or RedHat distributions, while retaining all the *coinstallability* information needed to determine if a set of packages can or cannot installed together. We refer the interested reader to [8] for precise figures and detailed proofs; we just recall here that this simplification is safe, and preserves all correct solutions.

3.2 Constraint generation

The second phase of `Zephyrus` translates the (trimmed) input into a set of constraints over non-negative integers. These constraints use different variables for the number of instances to create on each of the locations for each of the types in the universe, and also variables representing the packages that must be installed on each location. The constraints impose that the instances respect the definition of their type in the universe, the way how instances are implemented by packages, the (compacted) dependencies and conflicts between packages, and the problem specification.

The most interesting of these constraints ensure that it is possible to create the bindings between all instances according to the capacity constraints. These constraints distinguish our approach from others [13, 12, 14], they are necessary due to the flexible dependencies we have on ports. Constraints are constructed using auxiliary variables $B(p, t_r, t_p)$ for the number of bindings on port p between requesting instances of type t_r and providing instances of type t_p.

On the example of Section 2.2, these particular constraints for the bindings on port `sql` look like this:

$$B(\texttt{sql}, \texttt{wp}, \texttt{mysql}) \leq \texttt{\#mysql} * 3 \quad (1)$$

$$B(\texttt{sql}, \texttt{wp}, \texttt{mysql}) \geq \texttt{\#wp} * 2 \quad (2)$$

$$B(\texttt{sql}, \texttt{wp}, \texttt{mysql}) \leq \texttt{\#wp} * \texttt{\#mysql} \quad (3)$$

Here, (1) expresses that the number of bindings on port `sql` between instances of the two types is at most the number of instances of the providing type `mysql` times 3 (since any component of that type can bind to at most 3 instances), (2) that the number of bindings on port `sql` is at least the number of instances of the requesting type `wp` times 2 (the number of binding each component of type `wp` requires), and finally (3) states that the number of bindings is at most the number of pairs of instances of type `wp` with instances of type `sql`. This last restriction expresses that no two bindings may exist between the same pair of instances.

The ability to capture as simple integer constraints the existence of a complete architecture corresponding to a specification is the cornerstone of our approach. It allows us to deal with the many facets of a system design as a whole, and thus ensures the completeness of our tool and the optimality of the generated configuration. In particular, if the output of `Zephyrus` indicates that several services are to be installed on the same machine, we know that no conflict between the packages that realize them will arise on actual machines.

3.3 External solvers

The generated constraints, as well as the optimization function, are expressed using the `MiniZinc` constraint modelling language [24, 22]. This allows us to employ any of the many existing constraint solvers that support `MiniZinc`. `Zephyrus` can currently use GeCode [29] (an efficient open source solver) or several of the solvers provided in the G12 suite [30]. The tool exploits this flexibility by implementing a *solver portfolio* approach [2, 3] that reduces execution time by running several solvers in parallel, and stops as soon as one of the solvers finds a solution.

3.4 Configuration generation

When the external solver finds a solution for the generated constraint, the next part of `Zephyrus` proceeds to transform that solution, which is a simple mapping from variables to integers, into an actual configuration. The two main challenges of that generation are: i) to reuse as many existing parts of the initial configuration as possible in order to minimize the impact on the existing system; and ii) to correctly generate bindings between the instances taking into account that any two instances can be bound on a given port at most once. The algorithms employed in this generation phase are presented in detail in [7].

Once the configuration has been generated, it can be written to a file in two different formats: either (a) the same *JSON* format used for the input configuration, which precisely describes all the configuration features and is used by `Armonic` as input; or (b) the *dot* format that encodes the configuration into a graph that can be viewed using the `dot` program to visualize the synthesized architecture.

If no configuration satisfies the given input constraints, `Zephyrus` will exit with an error message and produce no output files.

3.5 Synthesis soundness and completeness

An important property of `Zephyrus` is that all its parts have been formalized. In particular, the translation into constraints and the generation of the configuration, two very complex and important pieces of `Zephyrus`, have been precisely described and proven correct in [7], where the following results have been shown for `Zephyrus`:

Theorem 1 (Soundness)**.** *The configuration generated by* `Zephyrus` *is correct w.r.t. the input universe and specification.*

Theorem 2 (Completeness)**.** *If there exists a configuration that validates the input universe and specification, then* `Zephyrus` *will successfully generate a correct configuration.*

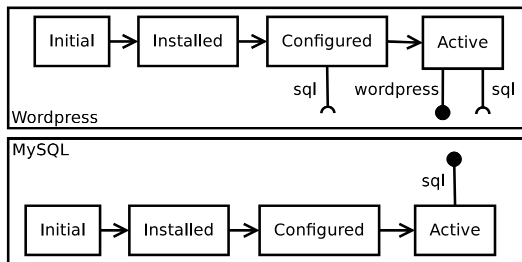


Figure 4: `Armonic` representation of Mysql and Wordpress component types.

Theorem 3 (Optimality). *The configuration generated by Zephyrus is optimal w.r.t. the input optimization function.*

3.6 Deployment planning

Virtual machine selection. In this paper, `Armonic` uses the popular OpenStack platform to provision virtual machines, starting from the *locations* entries found in `Zephyrus` output to determine machine names and resources (compute, storage, and memory capacity). In order to map resources to the available VM "flavors", a correspondence table is defined between `provide_resources` and Openstack flavors, for instance:

```
[["ram", 2048]] -> m1.small
```

Using this table, `Armonic` can create VMs using the Openstack API, e.g.:

```
nova boot --flavor m1.small --image debian-squeeze loc1
```

Component life-cycle. `Armonic` associates each component type to an acyclic state diagram that captures the component life-cycle. As an example, Figure 4 shows the life-cycles for the Worpdress and Mysql component types that we have used in the walkthrough of Section 2.

Different states may require and provide different ports. For instance, MySQL's active state provides a port `@sql`, denoting that such port can be depended upon only when that state in the MySQL life-cycle has been reached. Given that Wordpress' configured (and subsequent) state(s) require that port, Wordpress cannot enter such state before MySQL has entered its active state.

Determining deployment order. Section 2.3 briefly introduces the need of a component deployment plan. Computing such a plan can be quite challenging (even undecidable [9]), depending on the expressivity of component constraints. Hence, `Armonic` makes several simplifying assumptions to keep the problem tractable.

First, `Armonic` currently assumes that all VMs are "empty", with no services initially deployed on them. This assumption simplifies deployment of services because it ignores reconfiguration needs. Second, we suppose there is only one path to reach a given state, while `Armonic` life-cycle representation allows for multiple paths. Finally, we suppose that there are no circular dependencies between components. We are working on an improved planning algorithm which will address these limitations [18] and also allow for optimizations such as maximally parallel component deployment.

3.7 Service configuration

To connect components according to `Zephyrus` bindings, `Armonic` has to generate service configuration files and may have to create resources. For instance, connecting MySQL to Wordpress consists of creating a MySQL database and user with the appropriate permissions, and making a Wordpress configuration file point to the right IP address, port, DB name, and user name.

To automate this process, `Armonic` allows to attach additional information to provide and require ports. Thus, the provide port `@sql` of MySQL exposes three required variables, a database name, a user name, and a user password. The require port `@sql` of Wordpress exposes these variables with predefined values. Since the component MySQL is the provider of the bindings `@sql` which has Wordpress as requirer, `Armonic` uses this information to bind requirer and provider configuration variables. In this case, `Armonic` will call the provide port `@sql` of MySQL with values of require port `@sql` of Wordpress. This action will create the database and the MySQL user. These values will then be used by the `Armonic` agent to patch the Wordpress configuration file.

4. EXPERIMENTATION

The complete toolchain presented in this paper is available as free software, released under the GPL license. `Zephyrus` amounts to about 10.000 lines of OCaml [20] and is available at `https://github.com/aeolus-project/zephyrus/`; `Armonic` is about 5.000 lines of Python, plus glue code for component life-cycles written in shell script or Augeas [28], and is available at `https://github.com/armonic/armonic`. We have experimented the complete toolchain in both artificial and industrial settings; in this section we present some of our findings. As the figures for `Armonic` are dominated by the deployment time used by system-level tools (package managers, service startup, etc.), and also because we have already briefly presented them at the end of Section 2.3, we focus here on `Zephyrus`.

4.1 Synthesis efficiency

Given that the architecture synthesis part of our toolchain implemented by `Zephyrus` has a daunting complexity in theory one may ask whether this part could be a bottleneck of our approach. In order to answer this question we have conducted several architecture synthesis benchmarks on both realistic and extreme use cases. The ones we illustrate here are variants of the WordPress example described in Section 2. There are, however, three important changes needed to properly benchmark it:

- The use case is parameterized to scale it up and to demonstrate how `Zephyrus` handles problems which require more and more components and locations: (i) the first parameter is the minimum replication constraint on the `wordpress backend` port (required by the load balancer); (ii) the second parameter is the minimum replication constraint on the `sql` port (required by WordPress components).

- The resources associated to available locations are inspired by Amazon's EC2 VM offering. We took what

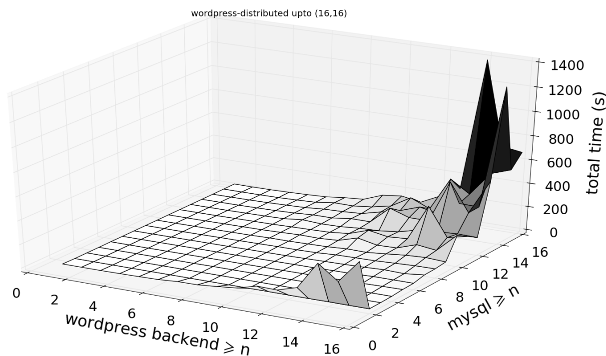


Figure 5: WordPress synthesis benchmarks, for increasing values of the replication constraints on wordpress backends (x-axis) and mysql (y-axis)

Amazon calls "old" (previous generation) general purpose machines. So there are four types of locations available, corresponding (by cost and capacity) to Amazon instance types: `m1.small`, `m1.medium`, `m1.large` and `m1.xlarge`. We have provided `Zephyrus` with a finite, but large enough number of machines (250 for each instance type).

- We use a single package repository associated to each machine, and a single package implementing all the available component types. This simplification does not affect the test results, as all component types in this benchmark are co-installable anyhow.

We have used a portfolio of solvers, as discussed in Section 3.3, consisting of the following 3 solvers: Gecode [29], the standard finite domain constraint solver from the G12 suite [30], and the G12/CPX (Constraint Programming with eXplanations) solver from the G12 suite. As these solvers are optimized for different goals, each of them works better in some situations and worse in others. It is very difficult to guess beforehand which solver is more adapted to a specific constraint problem instance. The portfolio approach permits us to work around this obstacle by trying these three approaches at the same time.

We have varied the two use case parameters from 1 to 16. Execution times are obtained as average of 5 runs on a commodity desktop machine (Intel i7 3.40 GHz, 8 GB of RAM). The diagram in Figure 5 shows that a vast majority of cases are solved very quickly in less than one minute. Only the larger ones can take more than 20 minutes, e.g. the `(14,16)` case, which is the highest peak in the chart. To put this worst-case solving time into perspective, please note that the largest use case (`(16,16)`) consists of 103 components, interconnected by 272 bindings, and distributed over 86 machines. This surpasses by a significant margin the size of most professional WordPress deployments.

4.2 Application to continuous integration

`Zephyrus` has been deployed in a large industrial use case at Kyriba Corporation[3], a large software editor providing Software-as-a-Service treasury management solutions. In the following we offer a return on that experience, validating the usefulness of the proposed approach in an industrial setting. This use case highlights the importance of considering all system design constraints together and the benefits of statically detecting when they are not satisfiable—in which case `Zephyrus` will exit with an error. It also shows the flexibility of our toolchain, by relying on a (in-house) deployment back-end other than `Armonic` for the actual deployment.

[3] `http://www.kyriba.com/`

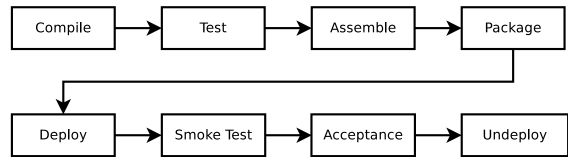


Figure 6: local qualification process at Kyriba

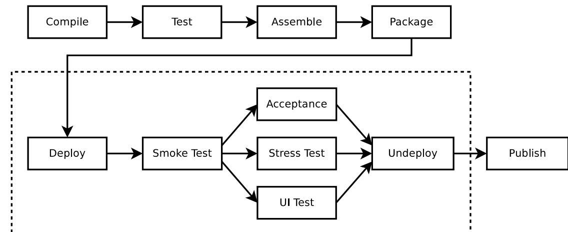


Figure 7: remote qualification process at Kyriba

Kyriba solution is a complex software platform composed of more than 150 components deployed on multi-tier architectures, with many different versions running at the same time. Maintaining the consistency of the system as a whole is a major undertaking. To address this challenge, Kyriba has invested in completely automating the build, integration, and deployment processes.

Kyriba distinguishes two software qualification processes: a local one run by individual developers on their machines; and another, more thorough one run on a remote continuous integration (CI) service. Heavy, exhaustive tests are performed remotely, whereas individual developers only run a subset of available tests on their machines.

Successful completion of the local qualification process, detailed in Figure 6, is required in order to be able to commit code changes to the source version control system. After each commit the process depicted in Figure 7 is triggered: first CI runs the same process that has been run on developers machine; automatic deployment is then performed on the cloud infrastructure with the latest component version, and more extensive tests—UI, deep functional scenarios, stress tests—are executed.

4.2.1 A Case for automation

Kyriba follows the continuous integration recommendations [10] and implements acceptance and stress tests. These tests are very time consuming: while local tests take less than 4 minutes to complete, global ones might take 4–8 hours. Furthermore, as *Kyriba solution* is an assembly of multiple components, integration tests involve many interdependent components that should all be deployed before testing. When deploying on a single machine, maintaining

consistency (e.g. version alignment) is rather easy and can be enforced using package dependencies; when components are distributed as services on multiple physical/virtual machines, consistency is harder to maintain.

In the past, test deployment was done using custom tools involving a manual setup, and component/protocol incompatibilities were only detected at runtime. Short feedback loops discipline helps developers with error diagnostic related to small code changes [16], so Kyriba has been looking for a tool that could anticipate error detection.

`Zephyrus` turned out to be a perfect fit for this need, as a deployment validation tool for both the local and remote qualification processes. `Zephyrus` is now used in distributed component consistency validation and deployment configuration scenarios. `Zephyrus` helps to get feedback before launching local deployments tests and has led to a significant reduction of the number of failures occurring during automated deployment in comparison to the previous, more manual, test setup.

4.2.2 Zephyrus adoption

For developers. Developers define relationships between components in partial `Zephyrus` files when they create packages for their project. These files are then merged with `Zephyrus` files containing the full infrastucture description provided by engineering operations team before being processed by the solver.

Two kinds of such relationships need to be defined:

- Dependencies between packages with specific version requirements, e.g. the application 1.0 requires a web server of version at least 3.2.4 to be installed on the same machine to run properly;

- Service binding relationship with API level requirement, e.g. the application 1.0 requires a service API version 1 exposed by another application on some machine, not necessarily where the application is deployed.

Developers can declare these requirements using ports specified in the `Zephyrus` universe definition:

```
{ "component_types": [
  { "name"  : "fa-accounting-engine-0.1",
    "provide": [["@fa-accounting-engine-v1", ["
     FiniteProvide", 1]]],
    "require": [["@graphite-v3", 1]] } ],
  "implementation": [
    [ "fa-accounting-engine-0.1",
      [["debian-kyriba",
        "kyriba-fa-accounting-engine (= 0.1)"]] ] ] }
```

The component type `kyriba-fa-accounting-engine` provides a “fa-accounting-engine-v1” service with API level 1 and requires a “graphite-v3” service. In the implementation section, the `component_type` is linked to the concrete package implementation `kyriba-fa-accounting-engine (= 0.1)`. This partial `universe` definition is merged with the full universe description (containing the definitions of all Kyriba components) and the default specification in order to verify that at least one final configuration exists. This ensures that no dependency problem can arise during deployment.

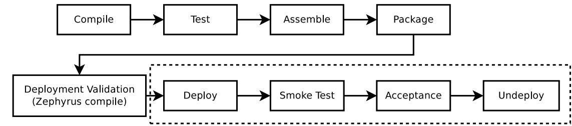


Figure 8: local qualification process with Zephyrus

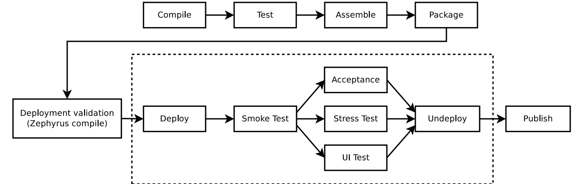


Figure 9: global qualification process with Zephyrus

Developers may also override the default `specification` with their own specification to validate different deployment scenarios during the local qualification process.

The local qualification process is modified by adding a `Zephyrus` validation stage before (local) deployment, see Figure 8. Using `Zephyrus` metadata developers simply declare component interfaces and the way they are exposed. Moreover, using `Zephyrus`, developers know beforehand if the components they are working on can be deployed together with other components.

For continuous integration and deployment environment. Similarly to the local one, the global qualification process has been instrumented with an extra `Zephyrus` validation stage as illustrated in Figure 9.

`Zephyrus` automatically checks application consistency before actually engaging in a deployment process. If the solver finds a solution, the related configuration is used by infrastructure management scripts based on the Fabric library[4] in order to orchestrate an integration test deployment on the cloud infrastructure (such as the Amazon Elastic Cloud Computing service). The newly created platform is then checked against all acceptance, stress and UI test cases.

To upgrade the production platform. `Zephyrus` is also used to plan platform upgrades on the infrastructure currently in production. According to the software road map, product managers define the component versions needed to be shipped to production for a milestone release. All those values are set in `Zephyrus` files, and based on the current production deployment, `Zephyrus` computes an output file containing the different application packages that should be installed with the related configuration parameters that need to be set for application binding. This guideline file is then used by engineering teams to write orchestration scripts and pinpoint manual upgrade tasks.

4.2.3 Outcome

Summing up, Kyriba’s experience with `Zephyrus` is that, instead of managing deployment scenarios manually using spreadsheets and flat documents, with *ad hoc* semantics leading to complex, time-consuming and error-prone deployments, `Zephyrus` brings precise semantics and simplifies the

[4] `http://www.fabfile.org`

automation of software qualification processes. Zephyrus provides static validation before actually performing expensive and very long dynamic validation at runtime. As most "compiler-like" tools, Zephyrus improves engineering quality and reduces building cost with less failures at deployment, integration test stage and platform upgrade.

5. RELATED WORK

The problem of managing networks of interconnected machines has attracted significant attention in the area of system administration. Many popular system management tools exist to that end: CFEngine [4], Puppet [17], MCollective [27], and Chef [26] are just a few among the most popular ones. Despite their differences, such tools allow to declare the components that should be installed on each machine, together with their configuration files. Then, they employ various mechanisms to deploy components accordingly. The burden of specifying where components should be deployed, and how to interconnect them is left to the sysadmin, let alone the difficult problem of optimal resource allocation. As an extra complication, system management tools stop at the package management abstraction, and therefore have no way of knowing *in advance* whether deployment will succeed or not. If the sysadmin requests to install two incompatible web servers on the same machine, the incompatibility will only be discovered by the package manager at deploy time, when one of the two services fails to get installed (or started). At that point, it is up to the admin to go back to the planning stage and work around the incompatibility. In our approach all package relationships are known to Zephyrus which can therefore plan around component incompatibilities.

System management tools can be used as an alternative to Armonic, though. Once optimal resource allocation is done by Zephyrus, the actual deployment can be delegated to them, now with the guarantee that no deployment error due to incompatibilities will arise; an interesting candidate could be [31].

CloudFoundry [32] specifically targets application deployment in the "cloud", but suffers from the same limitations as described above. ConfSolve [15] improves on the system management approach: it relies on a constraint solver to propose an optimal allocation of virtual machines to servers, and of applications to virtual machines, but it does not handle connections among services, nor capacity or replication constraints, and is unaware of package incompatibilities.

In Juju [5], each service is deployed on a single machine (or, more recently, in a virtual container on a machine). That avoids the issue of component incompatibilities, but does so at the price of wasting resources. In our Wordpress example Zephyrus proposes a solution that needs 4 machines, whereas Juju would have required 6.

Two recent efforts, Feinerer's work on UML [12] and Engage [14], are more similar to our approach as they both rely on a solver to plan deployments. Feinerer's work is based on the UML component model, which includes conflicts and dependencies with capacity constraints, but uses dependencies only between components, which greatly restricts the expressiveness of the model (choices are not possible). Moreover, no tool for actually building the computed configuration is provided. Engage, on the other hand, offers no support for conflicts in the specification language: one can only indicate that a service can be realized by exactly one out of a list of components. Neither Feinerer's work nor Engage, or any other tool that we are aware of, allows to find a deployment that uses resources in an optimal way, minimizing the number of needed (virtual) machines.

Another approach to automated deployment is proposed in [11], which uses an Architecture Description Language with user-provided information about relationships among software services, and implements a decentralized protocol to perform automatic configuration. This work may also be used as a backend for Zephyrus.

Finally, we would like to put Zephyrus in perspective. In our view, automated management of cloud applications is best realized by a 2-phase approach. In the first phase (*architecture synthesis*) Zephyrus or similar tools are used to devise an optimal system architecture. Then, in a second phase (*planning*), the obtained configuration is compared with that of the existing system to produce a detailed deployment plan that migrates the existing system to the desired one. To implement planning, the actual state of services and their life cycles (e.g. how do they pass from an inert "installed" state to an "up and running" one? do dependencies and conflicts change in the meantime?), ignored for the purpose of this paper, become relevant again. Even though planning has been shown to be undecidable in the most general case [18], promising progress has been made on automated planning for restricted cases, like planning in the presence of complex activation requirements that include circular dependencies, whereas giving up the possibility of expressing component conflicts [19].

6. CONCLUSION

We have introduced an automated approach to the design and deployment of complex distributed applications composed of interconnected services, as typically found in modern "cloud" environments. The system architect can specify the components needed to obtain the required functionalities, add non-functional constraints—e.g. maximum number of client components connected to a given service, or minimum number of replicas—as well as available physical resources—e.g. memory or bandwidth—and declare component incompatibilities. The architect can also choose an optimization goal, allowing to specify whether she prefers a conservative solution that changes the current configuration as little as possible, or a minimum-cost solution.

The approach is realized by two complementary tools: Zephyrus will synthesize an optimal architecture, including precise information about service interconnections. Such an architecture is then fed to the second tool, Armonic, which is able to deploy it on state-of-the art cloud infrastructures such as OpenStack, taking care of all deployment aspects from machine provisioning to service configuration and initialization. A major advantage of the proposed approach w.r.t. the state of the art is that all existing constraints, including software package-level incompatibilities, are taken into account to prevent deploy-time errors. We have validated the complete toolchain both theoretically, showing soundness and completeness of the approach, and practically by applying it to relevant industrial use cases.

To the best of our knowledge this toolchain is the first that allows to consistently handle capacity and replication constraints, conflicts, and service co-location, thus finally providing an instrument able to handle the stringent requirements of cloud applications in the real world.

7. REFERENCES

[1] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya. Software architecture optimization methods: A systematic literature review. *IEEE TSE*, 39(5):658–683, 2013.

[2] R. Amadini. Evaluation and application of portfolio approaches in constraint programming. *Theory and Practice of Logic Programming (TPLP)*, 13(4-5-Online-Supplement), 2013.

[3] R. Amadini, M. Gabbrielli, and J. Mauro. An empirical evaluation of portfolios approaches for solving CSPs. In C. P. Gomes and M. Sellmann, editors, *Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, volume 7874 of *LNCS*, pages 316–324, 2013.

[4] M. Burgess. A site configuration engine. *Computing Systems*, 8(2):309–337, 1995.

[5] Canonical Ltd. Juju, devops distilled. `https://juju.ubuntu.com/`. Retrieved October 2013.

[6] M. Catan, R. Di Cosmo, A. Eiche, T. A. Lascu, M. Lienhardt, J. Mauro, R. Treinen, S. Zacchiroli, G. Zavattaro, and J. Zwolakowski. Aeolus: Mastering the complexity of cloud application deployment. In *ESOCC 2013: Service-Oriented and Cloud Computing*, volume 8135 of *LNCS*, pages 1–3, 2013.

[7] R. Di Cosmo, M. Lienhardt, R. Treinen, S. Zacchiroli, and J. Zwolakowski. Optimal provisioning in the cloud. Technical report, Université Paris Diderot, 2013. Available at `http://hal.archives-ouvertes.fr/hal-00831455/`.

[8] R. Di Cosmo and J. Vouillon. On software component co-installability. In *Foundations of Software Engineering (FSE)*, pages 256–266. ACM, 2011.

[9] R. Di Cosmo, S. Zacchiroli, and G. Zavattaro. Towards a formal component model for the cloud. In *Software Engineering and Formal Methods (SEFM) 2012*, volume 7504 of *LNCS*, pages 156–171, 2012.

[10] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[11] X. Etchevers, T. Coupaye, F. Boyer, and N. de Palma. Self-configuration of distributed applications in the cloud. In *International Conference on Cloud Computing*, pages 668–675. IEEE, 2011.

[12] I. Feinerer. Efficient large-scale configuration via integer linear programming. *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing (AI EDAM)*, 27(1):37–49, 2013.

[13] I. Feinerer and G. Salzer. Consistency and minimality of UML class specifications with multiplicities and uniqueness constraints. In *Theoretical Aspects of Software Engineering (TASE)*, pages 411–420, 2007.

[14] J. Fischer, R. Majumdar, and S. Esmaeilsabzali. Engage: a deployment management system. In *PLDI'12: Programming Language Design and Implementation*, pages 263–274. ACM, 2012.

[15] J. A. Hewson, P. Anderson, and A. D. Gordon. A declarative approach to automated configuration. In *LISA '12: Large Installation System Administration Conference*, pages 51–66, 2012.

[16] J. Humble and D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[17] L. Kanies. Puppet: Next-generation configuration management. *;login: the USENIX magazine*, 31(1):19–25, 2006.

[18] T. A. Lascu, J. Mauro, and G. Zavattaro. Automatic component deployment in the presence of circular dependencies. In *Formal Aspects of Component Software (FACS) 2013*, volume 8348 of *LNCS*, 2013.

[19] T. A. Lascu, J. Mauro, and G. Zavattaro. A planning tool supporting the deployment of cloud applications. In *International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 213–220. IEEE, 2013.

[20] X. Leroy, D. Doligez, J. Garrigue, and D. Rémy. *The Objective Caml system release 4.01; Documentation and user's manual*. INRIA, Rocquencourt, Paris, 2013.

[21] F. Mancinelli, J. Boender, R. D. Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *International Conference on Automated Software Engineering (ASE)*, pages 199–208. IEEE, 2006.

[22] K. Marriott, N. Nethercote, R. Rafeh, P. J. Stuckey, M. G. de la Banda, and M. Wallace. The design of the zinc modelling language. *Constraints*, 13(3):229–267, 2008.

[23] I. Neamtiu and T. Dumitras. Cloud software upgrades: Challenges and opportunities. In *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2011 International Workshop on the*, pages 1–10, Sept. 2011.

[24] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming (CP)*, pages 529–543, 2007.

[25] Normation. Rudder – open source automation & compliance. `https://www.rudder-project.org/`. Retrieved April 2014.

[26] Opscode. Chef. `http://www.opscode.com/chef/`. Retrieved October 2013.

[27] Puppet Labs. Marionette collective. `http://docs.puppetlabs.com/mcollective/`. Retrieved October 2013.

[28] RedHat. Augeas – a configuration API. `http://augeas.net/`. Retrieved April 2014.

[29] C. Schulte, M. Lagerkvist, and G. Tack. Gecode. `http://www.gecode.org/`. Retrieved October 2013.

[30] P. J. Stuckey, M. G. de la Banda, M. Maher, J. Slaney, Z. Somogyi, M. Wallace, and T. Walsh. The G12 project: Mapping solver independent models to efficient solutions. In *International Conference on Logic Programming (ICLP)*, volume 3668 of *LNCS*, pages 9–13, 2005.

[31] S. van der Burg, E. Dolstra, and M. de Jonge. Atomic upgrading of distributed systems. In *Hot Topics in Software Upgrades*, pages 1–5. ACM, 2008.

[32] VMWare. Cloud Foundry, deploy & scale your applications in seconds. Retrieved October 2013, `http://www.cloudfoundry.com/`.

CHAPTER 12

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

This chapter contains the full text of the article “Debsources: Live and Historical Views on Macro-Level Software Evolution” *[31].*

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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEM'14 September 18–19, 2014, Torino, Italy.
Copyright 2014 ACM 978-1-4503-2774-9/14/09 ...$15.00.

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

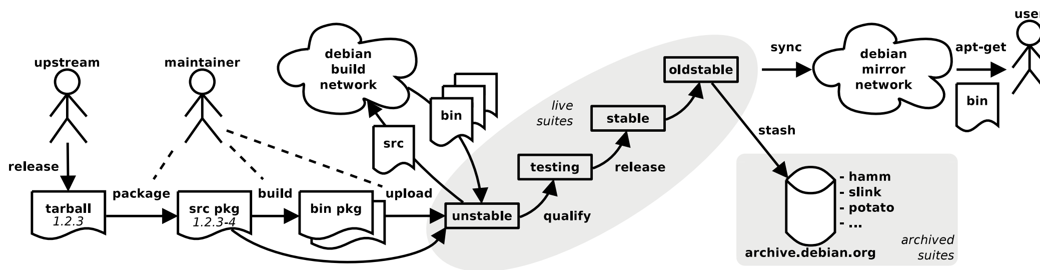


Figure 1: Life-cycle of Debian packages and releases

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

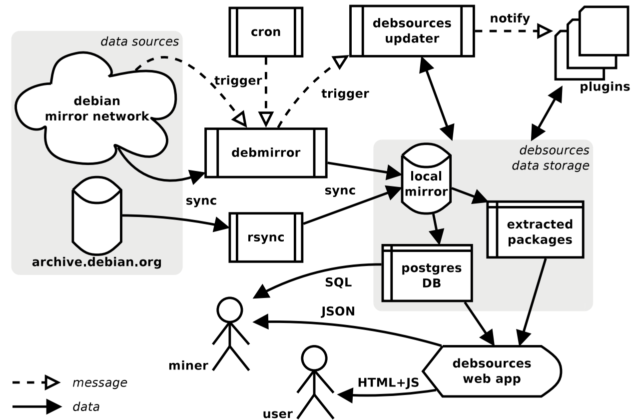


Figure 3: Debsources architecture

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`

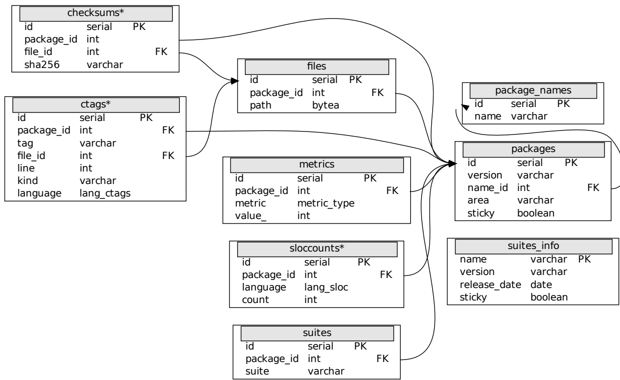


Figure 4: Debsources DB schema (excerpt); * denotes tables pertaining to plugins

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.

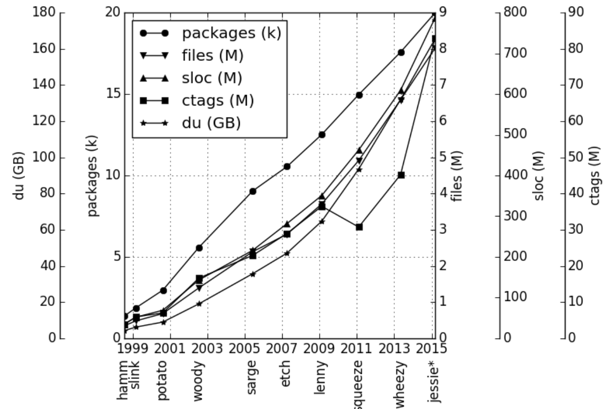


Figure 5: Debian release sizes over time

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

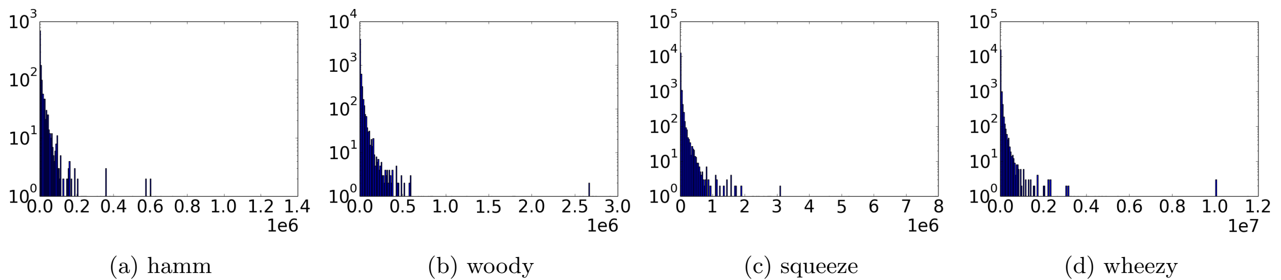
Long-term evolution trends do not seem to be affected by these differences though. Before *etch* (last release considered in ref. study) both SLOC and package counts grow linearly with time and super-linearly with releases. Interestingly, post-*etch* the growth rate has increased and is now super-linear 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*

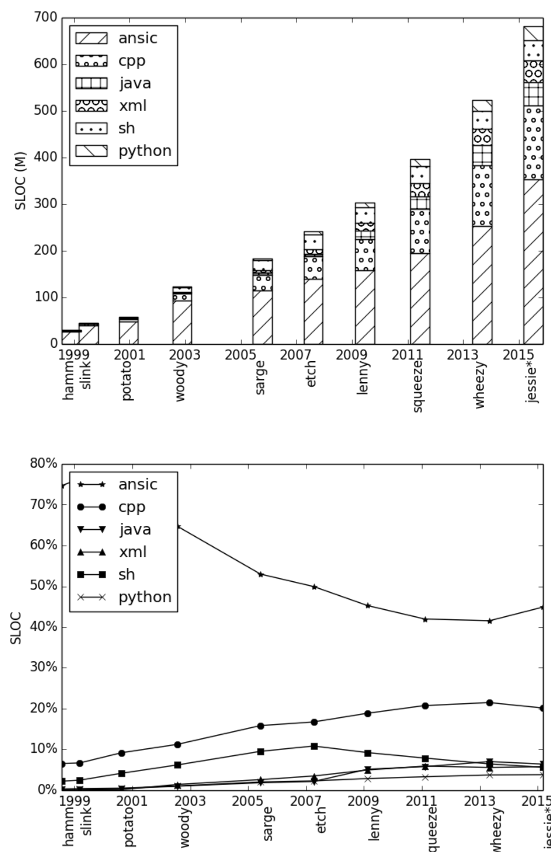


Figure 7: most popular (top-5) programming languages in Debian over time

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.

CHAPTER 13

The Debsources Dataset: Two Decades of Free and Open Source Software

This chapter contains the full text of the article "The Debsources Dataset: Two Decades of Free and Open Source Software" *[30].*

Noname manuscript No.
(will be inserted by the editor)

The Debsources Dataset: Two Decades of Free and Open Source Software

Matthieu Caneill · Daniel M. Germán · Stefano Zacchiroli

Received: date / Accepted: date

Abstract We present the Debsources Dataset: source code and related metadata spanning two decades of Free and Open Source Software (FOSS) history, seen through the lens of the Debian distribution.

The dataset spans more than 3 billion lines of source code as well as metadata about them such as: size metrics (lines of code, disk usage), developer-defined symbols (ctags), file-level checksums (SHA1, SHA256, TLSH), file media types (MIME), release information (which version of which package containing which source code files has been released when), and license information (GPL, BSD, etc).

The Debsources Dataset comes as a set of tarballs containing deduplicated unique source code files organized by their SHA1 checksums (the source code), plus a portable PostgreSQL database dump (the metadata).

A case study is run to show how the Debsources Dataset can be used to easily and efficiently instrument very long-term analyses of the evolution of Debian from various angles (size, granularity, licensing, etc.), getting a grasp of major FOSS trends of the past two decades.

The Debsources Dataset is Open Data, released under the terms of the CC BY-SA 4.0 license, and available for download from Zenodo with DOI reference 10.5281/zenodo.61089.

This work has been partially performed at IRILL, center for Free Software Research and Innovation in Paris, France http://www.irill.org. Unless noted otherwise, all URLs in the text have been retrieved on September 1st, 2016. Authors are listed alphabetically.

M. Caneill
Université Grenoble Alpes, Grenoble, France, E-mail: caneill@imag.fr

D. Germán
University of Victoria, Victoria, Canada, E-mail: dmg@uvic.ca

S. Zacchiroli
Univ Paris Diderot, Sorbonne Paris Cité, IRIF, UMR 8243, CNRS, F-75205 Paris, France, and Inria, France, E-mail: zack@pps.univ-paris-diderot.fr

1 Introduction

Software is increasingly being distributed to final users by the means of software collections and deployed using package management tools. Some software collections are very tightly curated and integrated, like Free and Open Source Software (FOSS) *distributions*, others much more loosely so, like so called "app stores". The study of software evolution [19, 16] can no longer ignore software collections as relevant subjects of macro-level studies [11, 3], i.e., evolution studies conducted at the granularity of component releases rather than individual commits.

The study of software collections however poses specific challenges to scholars due to a common tendency at creating *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.

The Debsources platform [3] has been developed to counter those challenges in the specific case of Debian[1]—a general purpose FOSS operating system for desktop and server computers, which is one of the most reputed and oldest (est. 1993) distributions, often credited as the largest curated collection of FOSS components. Thanks to the free availability of both its source code and associated metadata, its conspicuous size, and its standardized package layout [13], Debian has become a popular subject of empirical software engineering studies [4] (see, among many others, [25, 11, 1]).

Debsources allows to gather, index, search, and publish on the Web the entire source code of Debian and metadata extracted from it. The most notable instance of Debsources is publicly available at http://sources.debian.net and indexes all currently active Debian releases, with several updates per day, as well as historical Debian releases going back almost 20 years.

Contributions In this paper we present the Debsources Dataset, a polished version of the data underpinning http://sources.debian.net suitable for a wide range of large-scale analyses on FOSS components released by Debian. The dataset is composed of two parts that can be used together or independently:

1. **Source code**. The dataset includes the source code of 10 Debian stable releases published over the past 2 decades, corresponding to 82 thousand packages for more than 30 million source code files. To reduce storage size, source code files have been deduplicated and organized in a manner that facilitates and speeds up empirical studies. The result of deduplication is 15 million unique files, requiring ≈ 320GB of disk space. After compression with `xz` the source code part of the dataset shrinks down to ≈ 90GB.
2. **Metadata**. Rich metadata regarding all shipped source code are also part of the dataset. Release metadata link together the 10 Debian releases, the packages that compose each of them, and the source code files that form

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

each package. In addition to release information, the dataset also contains the following content-oriented metadata:

- per-file size metrics including file size in bytes, number of lines (using `wc`), source lines of code (SLOC) divided by language, computed using both `sloccount`,[2] and `cloc`;[3]
- checksums: cryptographic hashes SHA1 and SHA256, as well as locality-sensitive TLSH [18] hashes of all files;
- MIME media type of each file, as detected by `file`[4];
- location, name, and type of developer-defined symbols (functions, data types, classes, methods, etc.) obtained indexing all source code with Exuberant Ctags;[5]
- applicable FOSS license for individual files, as detected by both `ninka` [8] and `fossology` [9].

Source code is shipped as a set of tarballs, metadata as a PostgreSQL[6] database dump.

Exploitation ideas The Debsources Dataset is a valuable resource for scholars interested in studying either the composition or the long-term evolution of FOSS. Here are a few ideas—some already realized, some still up for grabs—on how to exploit the dataset:

- Conduct or replicate long-term, macro-level *evolution studies* of FOSS. We show an example of this in Section 5.
  Several followup research questions remains unanswered, e.g.: does the use of different programming languages evolve in similar ways along the history of development? 20 years are enough to observe the raise and fall of programming languages and try to spot interesting adoption patterns. Using the Debsources Dataset those studies can be done both in aggregate ways (e.g., how many software projects are written in a given language over time?) and at per-project level (e.g., do all software projects written in a given language follow similar evolution patterns?).
- Study the structure and evolution of *license use* in FOSS, at different granularities: file, package, distribution. We address some of these in Section 5, but a lot remains to be done, most notably in the area of licensing of software components as aggregate wholes.
- Investigate *code reuse and cloning* along the whole history of all software packages contained in the dataset. Reuse without modification is trivial to track thanks to SHA1 and SHA256 checksums. Reuse *with* modification can be supported using ctags and/or TLSH hashes as fingerprinting techniques to track (modified) code copies, or by directly parsing the actual source code available in the dataset.

[2] http://www.dwheeler.com/sloccount/
[3] https://github.com/AlDanial/cloc
[4] http://www.darwinsys.com/file/
[5] http://ctags.sourceforge.net/
[6] http://www.postgresql.org

– The availability of source code can be further leveraged to support several kinds of static analysis studies. By focusing on source code files written in a specific programming language (e.g., C, C++), researchers can study the evolution over time of bugs that are detectable with a given static analysis tool (e.g., Coccinelle, Coverity).
  On the more practical side, the source code in the dataset also forms an interesting benchmark for *code search* at a scale. Multi-language, license-aware, automatic code completion backed by the Debsources Dataset would make for a very fun and useful toy for many developers.
– In comparison with other sub-fields, *release engineering* [2] is still relatively unexplored in empirical software engineering. The Debsources Dataset allows to follow the evolution of package-level structures along 20 years of Debian, and to mix-and-match with release metadata, metrics, and actual source code. Some open research questions in this area are: when and how software projects get split into multiple packages? does package organization change over time? does that affect release schedules? how do packages migrate from one development release to another? etc.

Paper structure Section 2 explains how the Debsources Dataset has been assembled. Section 3 describes the data schema and gives some statistics about its content. Section 4 shows how to get started using the dataset. Sections 5 presents a case study on how the dataset can be used to conduct a long-term, macro-level evolution study of FOSS from several angles using Debian as a proxy. Section 6 discusses the limitations of the dataset. Before concluding, Section 7 points to related work.

Dataset availability The Debsources Dataset is Open Data. The metadata part of the dataset is available under the terms of the Creative Commons Attribution-ShareAlike (CC BY-SA) license, version 4.0; the source code part of the dataset is available under the terms of the applicable FOSS licenses. The dataset is available for download from Zenodo[7] at https://zenodo.org/record/61089, with DOI reference 10.5281/zenodo.61089.

2 Data gathering

The Debsources Dataset has been assembled by mirroring and extracting Debian source releases, organizing extracted source code to remove duplicates, running analysis tools over the obtained files, and injecting their results in a PostgreSQL database.

Given that both all Debian releases and the analysis tools we have used are freely available as FOSS, the dataset can be recreated from scratch following the blueprint given below. Be warned though that (re-)creating the dataset takes a significant amount of resources, both in terms of processing time and

[7] https://zenodo.org/

required disk space; details are given below. Also note that to simply *use* the Debsources Dataset you do *not* need to go through the process below, which is documented here for information and reproducibility purposes only. The dataset is ready to use as-is; see Section 4 for a quick start guide.

2.1 Blueprint

Database structure

0. As a preliminary step we have created the database structure. This can be achieved by replaying in a freshly created database the schema creation SQL statements that can be found in the Debsources Dataset database dump. The database content has then been filled as we went during the process described below.

Mirror current and historical Debian releases

1. *Current releases.* We have used debmirror[8] to retrieve all current Debian releases from a nearby mirror.[9] Binary packages can be ignored, and one can easily tune debmirror to only download source packages.
2. *Historical releases.* We have used `rsync` [23] to mirror `http://archive.debian.org`. This step is required to retrieve historical Debian releases that are no longer available from the regular mirror network.
3. *Releases metadata.* To load into the database release information we have used various sources of information. Each Debian release comes with a set of `Sources` files describing which packages/versions compose the release as well as other package metadata. We have parsed those files using the `python-debian` library[10] and stored the extracted information in the database.
   For the static information about each release (release name, date, etc.) we started from the list of Debian releases on Wikipedia,[11] double-checked with official Debian release announcements, and stored the obtained information in the database.

Source code extraction and deduplication

4. *Extract packages.* Debian source packages are formed by one or more tarballs and/or patch sets, along with a `.dsc` manifest file. We have looked for all such manifest files and extracted their content using `dpkg-source -x package_version.dsc` (dpkg-source is a Debian tool that can be found in the `dpkg-dev` package). Doing so merges upstream tarballs together

[8] `https://packages.debian.org/sid/debmirror`

[9] A list of Debian mirrors organized by geographical location is available at `https://www.debian.org/mirror/list`.

[10] `https://packages.debian.org/sid/python-debian`

[11] `https://en.wikipedia.org/wiki/List_of_Debian_releases`

and apply Debian-specific patches in the process. Thanks to the unicity of package ⟨name, version⟩ pairs, all packages can be extracted in the same directory; each one will be extracted in a separate sub-directory without conflicts.

5. *Deduplicate files.* Due to the presence of multiple versions of the same software, many source code files are exact copies of others. By deduplicating identical files we have cut down both disk usage and the number of files to process for any kind of batch analysis by a factor 2. To this end:
   - We have listed all regular files (e.g., with `find -type f`). This step excludes symbolic links, which can also result in processing multiple times the same files.
   - For each regular file, we have computed its SHA1 checksum and stored it in the database.
   - For each unique SHA1, we have created a file `XX/YY/SHA1`, whose content is identical to the original file and where `XXYY` are the first 4 characters of the checksum.
   - *Mirror deduplicated files to preserve extensions.* Some file analysis tools require file extensions to properly identify the file language, type, or format. For example, without a `.c` extension many C files will not be recognized as such by default by neither `cloc` nor `sloccount`. To facilitate the analysis with such tools we have created a directory tree with the same structure of the deduplicated tree above, but having extensionful symlinks as its leaves. Each symlink is named `XX/YY/SHA1.ext` and points to `XX/YY/SHA` in the deduplicated tree.

Compute content metadata

6. *Extension-agnostic metadata.* For each unique SHA1, we have retrieved the corresponding unique file and computed its size, media type (using the Unix command `file`), number of lines (using the Unix command `wc`), SHA256, and TLSH checksums.
7. *Extension-sensitive metadata.* For each unique ⟨SHA1, extension⟩ pair, we have ran `cloc`,[12] `sloccount`,[13] and exuberant `ctags`[14] on the corresponding extensionful symlink. Note that all these operations can also be performed in batch on several files at once. For instance, one can execute `ctags --recurse` on the outermost `00` SHA1 directory, to process all files whose SHA1 starts with `00` at once. The trade-off here is between the output size of each tool invocation and the number of invocations.

   All obtained metadata have been stored in the database.

Compute license information

8. *License detection.* We ran `fossology` and `ninka` on all unique files and stored the output of both license detection tools in the database.

[12] https://github.com/AlDanial/cloc
[13] http://www.dwheeler.com/sloccount/
[14] http://ctags.sourceforge.net/

2.2 Required resources

The dataset (re-)creation process is I/O-bound and might require up to 1.5TB of working disk space during processing.

About 200GB are needed to store the (compressed) source mirror of both current and historical Debian releases. After extraction, but before deduplication, 800GB of additional disk space are required to store the bulk of the extracted source code. Deduplication and release metadata extraction can then be run, resulting in extra 400GB between the new files and the working database. At this point the 1TB of disk space occupied by compressed and uncompressed source code can be freed.

Processing the deduplicated source code to compute checksums, SLOCs using `sloccount`, `ctags`, and disk usage took us about 10 days on a single server-grade machine with rather slow (by today standards) 7.2kRPM spinning disks. Computing SHA256, TLSH, media types, and SLOCs using `cloc` required approximately 1 week using an Apple Mac Pro and a Promise RAID. The process of running `fossology` and `ninka` on all source files took approximately 25 days on a virtual machine using the Western Canada Research Grid.[15]

In total, a realistic estimate for recreating from scratch the Debsources Dataset on a single machine equipped with fast SSD drives is in between 4 and 5 weeks; a couple of weeks more with spinning disks.

3 Bird's eye view

The Debsources Dataset comes in two major parts: a set of tar files containing the source code and a PostgreSQL database dump with metadata about source code files and their relationship to Debian packages and releases.

3.1 Source code

The first part of the Debsources Dataset is a set of tarballs containing the deduplicated source code files. They were divided into 16 tarball (each one weighting 5–6GB) to facilitate distribution. Each tarball, named `debsources.`*X*`.tar.xz` (where X is an hexadecimal digit: 0–9, a–f), includes all source code files whose SHA1 start with X. The files contained in these tarballs are further divided (or "sharded") in sub-directories based on the first 4 characters of their SHA1. For example, a file whose SHA1 is `deadbeef[...]` will have path `de/ad/deadbeef[...]` and can be found in tarball number `d`.

As discussed in Section 2, the additional tarball `debsources-ext.tar.xz` contains file extension information as a set of symlinks to the actual source code files.

[15] https://www.westgrid.ca/

Table 1 Various versions of the `xournal` package in Debian, all containing a file named `src/xo-interface.h` with the same SHA1 checksum.

Release	Package	Version	SHA1 of `src/xo-interface.h`
lenny	xournal	0.4.2.1-0.1	e09a07941a3c92140c994fcdda7f74bce1af4ca3
squeeze	xournal	0.4.5-2	e09a07941a3c92140c994fcdda7f74bce1af4ca3
wheezy	xournal	0.4.6~pre20110721-1	e09a07941a3c92140c994fcdda7f74bce1af4ca3
jessie	xournal	1:0.4.8-1	e09a07941a3c92140c994fcdda7f74bce1af4ca3

Consider file `src/xo-interface.h`, which can be found in four different versions of the `xournal` package in Debian. Its metadata are depicted in Table 1. After extraction (see Section 4) source code can be found in two top-level directories: `debsources` and `debsources.ext`. The first one contains the actual source code, the second extensionful symlinks to them. In our example we will have the following on-disk layout (names ending in '`/`' represent directories, and '`->`' a symlink and its destination):

```
debsources/
...
debsources/e0/
debsources/e0/9a/
debsources/e0/9a/e09a07941a3c92140c994fcdda7f74bce1af4ca3
...
debsources.ext/
...
debsources.ext/e0/
debsources.ext/e0/9a/
debsources.ext/e0/9a/e09a07941a3c92140c994fcdda7f74bce1af4ca3.h ->
   ../../../sources/e0/9a/e09a07941a3c92140c994fcdda7f74bce1af4ca3
...
```

Even though files are renamed to match their SHA1 checksums, the directory structure of individual packages is not lost. Full original paths are available as part of the metadata database described next. As a preview, the following SQL query can be used to reconstruct the paths at which a file with the SHA1 of our example can be found in the dataset, together with the corresponding package names and versions:

```
SELECT release_id as release,
       package_name as package, package_version as version,
       encode(path, 'escape') as path
FROM releases
NATURAL JOIN package_info
NATURAL JOIN paths
NATURAL JOIN path_info
NATURAL JOIN files
WHERE sha1='e09a07941a3c92140c994fcdda7f74bce1af4ca3'
```

When run the query will return the following tuples:

release	package	version	path
lenny	xournal	0.4.2.1-0.1	src/xo-interface.h
squeeze	xournal	0.4.5-2	src/xo-interface.h
wheezy	xournal	0.4.6~pre20110721-1	src/xo-interface.h
jessie	xournal	1:0.4.8-1	src/xo-interface.h

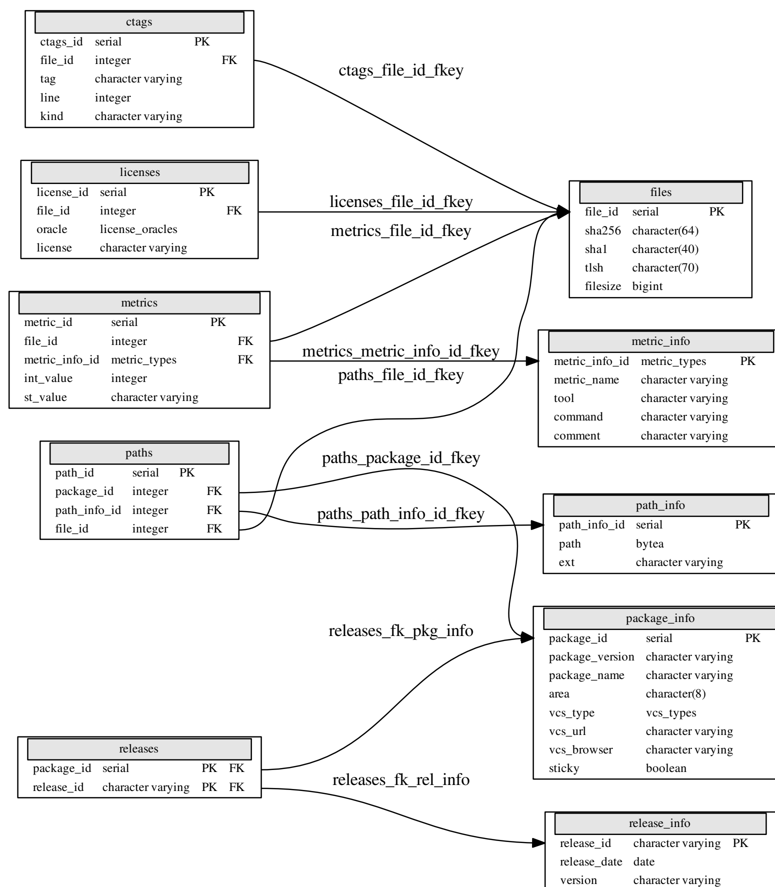


Fig. 1 Database schema. Primary key fields are denoted with "PK", foreign keys "FK"; arrows indicate referential integrity constraints.

3.2 Metadata

The second part of the dataset is a Postgres database, containing all source code metadata. The database schema is shown in Fig. 1. A brief description of each table is given below.

3.2.1 Intrinsic information

The following tables describe releases, packages, files and the relationships among them:

Table 2 Tools used to extract file-level metadata.

Tool	Version	Command
file	5.14	`file --mime-type`
cloc	1.66	`cloc --by-file --follow-links --skip-uniqueness \ --sql-append`
sloccount	2.26	`sloccount --duplicates --follow --details`
wc (coreutils)	8.13	`wc -l`

– *package_info:* information about Debian source packages contained in the dataset, such as package names, versions, and associated attributes (e.g., project homepage). Package names are generally lowercase variants of the original (or "upstream") FOSS project names, e.g., bash, linux (the kernel), libreoffice, etc. Package versions include both the upstream project version and the Debian package revision separated by a dash, e.g.: "1.2.3-4".
– *release_info:* information about the 10 Debian releases in the dataset; name, version, and release date of each one are included.
– *releases:* mappings between source packages and Debian releases.
– *path_info:* the full path name of every file in the dataset. The table also contains, as a separate field, the file extension. Note that, consistently with the POSIX standard, paths are stored as raw byte sequences; there is no guarantee that they can be interpreted as valid Unicode characters without further knowledge of the applicable character encoding. The vast majority of paths that are encoded in UTF-8 can be parsed at query time using suitable Postgres functions.
– *files:* deduplicated files together with their checksums (SHA1, SHA256, and TLSH) and size (in bytes). This table lists all unique files present in the dataset.
– *paths:* ternary mappings between *packages*, *path_info*, and *files*. This table indicates, for a given package and path, the corresponding unique file. For symbolic links, the *file_id* column will be NULL.

3.2.2 Derived information

The following tables describe information that have been extracted from Debian source code following the process described in Section 2.

– *licenses:* license information. This table maps unique files to the corresponding FOSS licenses as identified by the license detection tools (or "oracles") `ninka` and `fossology`.
– *metric_info:* information about the tools used to compute file-level information. For reproducibility reasons, this table includes tool name, version, command line used to run it, and a *comment* field with additional human-readable information. Due to how representative of the available file-level information this table is, its content is given in Table 2.
– *metrics:* links files to the information extracted from them. Some derived information are integer-valued (e.g., the output of `wc -l`), some string-valued (e.g., `file` output), some both (`cloc` and `sloccount` output both

Table 3 Size of Debsources Dataset metadata as a Postgres database. The entire database requires ≈ 40GB of disk space (including indexes, which are not listed below).

Table	Disk size	Tuples
ctags	23 GB	186.5M
files	5944 MB	15.5M
metrics	3549 MB	46.7M
paths	3259 MB	30.5M
licenses	2976 MB	31.0M
path_info	1895 MB	11.7M
package_info	14 MB	82113
releases	7248 KB	97471
metric_info	32 KB	4
release_info	32 KB	10

Table 4 Size of Debsources Dataset source code.

Tarball	Disk usage (compressed)	Disk usage (expanded)
`debsources.*.tar.xz`	89GB (total)	317GB
`debsources-ext.tar.xz`	422MB	61GB
`debsources.dump.xz`	3.1GB	see Table 3

detected language and number of SLOCs). For this reason this table allows to store an integer (field `int_value`) and/or a string (`st_value`). The attribute `comment` in table *metrics_info* documents which field is relevant for which metric.

– *ctags:* `ctags` results for each file. The table contains one entry for each developer-defined symbols in a given source file, together with the precise file location at which the symbol was found and the symbol type (function, data type, method, etc).

3.3 Dataset size

To give an idea of the size of the dataset, Table 3 lists the sizes of all tables in the database, as both number of tuples and required disk space. If space is at a premium, some large tables (e.g., *ctags*) can be deleted without compromising the referential integrity of the database. Similarly, Table 4 details the required disk space to locally host the source code part of the Debsources Dataset.

4 Getting started

This section describes the steps necessary to use the Debsources Dataset. The two parts—source code and metadata—can be used independently, but the metadata are needed if you want to be able to relate individual source code files to their context.

4.1 Metadata

The metadata part of the Debsources Dataset comes as a plain-text SQL dump of a PostgreSQL database, compressed in `xz` format. The dump has been obtained from Postgres 9.4 using `pg_dump`, but it should be compatible with any version of Postgres $\geq$ 9.1.

To import the metadata you should first install Postgres, then create a dedicated database (e.g., *debsources*), and finally import the dump into it. For the last two steps you can proceed as follows, acting as a user with suitable Postgres permissions:

1. `createdb debsources`
2. `xzcat debsources.dump.xz | psql debsources`

On a modern high-end laptop equipped with a fast SSD disk, the import takes about 1.5 hours. The freshly imported database will require about 40 GB of disk space (see Table 3 for details).

4.2 Source code

To decompress the source code you should first create a directory that will contain all of it and move into that directory. Then:

– (Optional) Expand the tarball with the extension symlinks:
  `tar xaf /path/to/compressed/dataset/debsources-ext.tar.xz`
– Create a sub-directory called `debsources` and move into it:
  1. `mkdir debsources`
  2. `cd debsources`
– Extract the actual source code. For each of the `debsources.`*X*`.tar.xz` tarballs, execute:
  `tar xaf /path/to/compressed/dataset/debsources.X.tar.xz`

The result will be a Debsources Dataset directory containing two sub-directories, `debsources` for extensionless deduplicated source code files sharded by SHA1, and `debsources.ext` with extensionful symbolic links pointing into it, as described in Section 3.1.

One advantage of the deduplicated files is that they are file system agnostic and can be expanded onto any file system. This is not the case for the original source code files. For example, these two paths `net/netfilter/xt_tcpmss.c` and `net/netfilter/xt_TCPMSS.c` exist in the Linux kernel. The two paths point to files with different content, but have file names that differ only in capitalization. This is supported by file systems such as ext4, but will cause a file name clash on JFS (case preserving, Mac OS) or NTFS (while the file system is case sensitive *per se*, case sensitivity depends on the application creating the files). The impact of these low-level issue can be significant: there are more than twenty such cases in the Linux kernel alone. By renaming the files to their SHA1 we avoid similar issues.

5 Case study: long-term macro-level evolution

In the following we show how the Debsources Dataset can be used to conduct a long-term, macro-level evolution analysis of FOSS projects, as they can be observed through the lens of the Debian distribution. We focus on aspects such as source code size (under various metrics), programming language popularity, package size, package maintenance, and software licensing.

The analyses we conduct are both qualitative and quantitative, and in part replicate and extend previous findings [11, 3]. The research questions we will address are:

- RQ i. *How does the size of Debian evolve over time?* Looking at various metrics we will study how and at which rate Debian grows across releases.
- RQ ii. *How much Debian changes between releases?* By studying package versions and their content, we can measure the amount of packages that are updated across Debian releases and to what extent they are.
- RQ iii. *How has the popularity of programming languages changed over the last 20 years?* By looking at the evolution of SLOCs per language, we identify which languages are gaining (or losing) traction among FOSS projects represented in Debian.
- RQ iv. *Which licenses apply to Debian source code files?* We identify which software licenses are used in Debian at a file-by-file granularity, irrespectively of the containing package.
- RQ v. *Which licenses can be found in Debian source packages?* By aggregating file licenses by package we can study the expected license variability when reusing entire software packages.
- RQ vi. *How has license use evolved in Debian over time?* We explore the evolution of license use over time by comparing the licensing of files and packages that belong to different Debian releases.

5.1 Growth over time

The evolution of Debian size over time (RQ i) can be studied under various metrics. We take into account the following ones: number of packages, number of source code files, disk usage of (uncompressed) source code, lines of code (SLOCs), and developer-defined symbols (or "*ctags*").

In the Debsources Dataset packages can be found in the *package_info* table, that has one row per package. Source code files can be found in table *paths*, which in turn points to unique files listed in table *files*. All file-level metrics, except ctags, are in table *metrics*, column *int_value*, distinguished by metric type (column *metric_info_id*).[16] File-level metrics can then be grouped by package following the *metrics* $\rightarrow$ *files* $\leftrightarrow$ *paths* $\rightarrow$ *package_info* chain of

[16] Note that two different SLOC metrics are available in the dataset: as computed by `sloccount` and `cloc`. Each tool has its strength and weaknesses. For this case study we use `sloccount` numbers.

relationships. Ctags are stored in the separate *ctags* table because, whereas they can be used as a size/complexity metric for individual source code files, they primarily act as an index which doesn't fit the general model of the *metrics* table. Per-packages metrics can be further aggregated by release using the *releases* table. Per-release metrics can finally be sorted by time using the *release_date* field of the *release_info* table.

Table 5 Debian release sizes by various metrics—number of packages, files (and files explicitly recognized as source code by `sloccount`), disk usage of uncompressed source packages, lines of code, developer-defined symbols (ctags). See also Table 6 for additional statistics parameters about these measures.

Release	Version	Packages	Files (k)	Source files (k)	Disk usage (GB)	ctags (M)	SLOCs (M)
hamm	2.0	1373	348.4	152.5	4.1	4.1	34.9
slink	2.1	1880	484.6	224.4	6.0	6.2	51.9
potato	2.2	2962	686.0	292.6	8.6	7.4	68.8
woody	3.0	5583	1394.5	563.3	18.2	17.2	140.7
sarge	3.1	9050	2394.0	870.6	34.1	24.2	210.1
etch	4.0	10 550	2879.7	1092.7	45.0	30.3	272.1
lenny	5.0	12 517	3713.9	1437.2	61.8	38.3	332.7
squeeze	6.0	14 951	4908.1	1952.2	89.1	52.3	444.4
wheezy	7	17 564	7310.5	2751.4	131.7	69.5	636.8
jessie	8	21 041	8375.0	3404.2	167.0	95.6	784.3

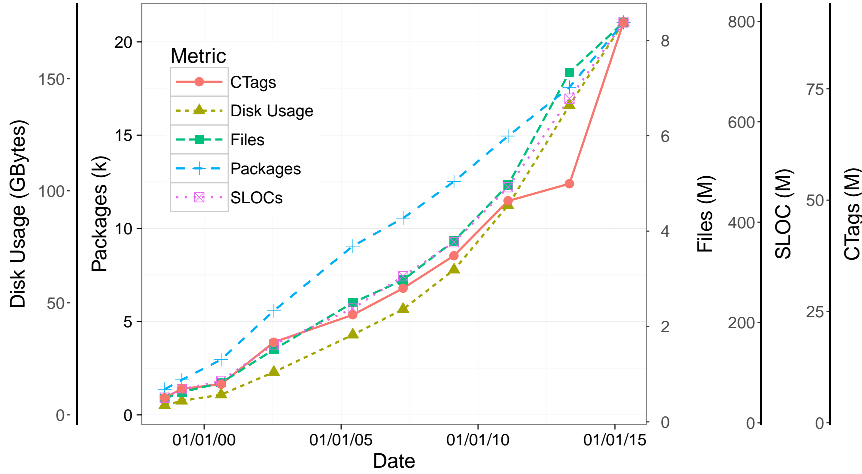


Fig. 2 Debian release size over time, under various metrics.

Release size The above query plan can easily be translated to SQL queries and run on the Debsources Dataset. Query results are shown in Table 5, and plotted in Fig. 2 over time.

In absolute terms, Debian has scaled to a point where the last stable release (Jessie) contains more than 21 thousand packages, and almost 800 millions lines of code. If we look at the metrics evolution over time we notice that the five considered metrics exhibit similar growth rates. Four of them (ctags, disk usage, files, and SLOCs) are very highly correlated and grow super-linearly, with an apparent slow down in the most recent stable release. The other metric (package count) is more regular and almost perfectly linear.

This discrepancy gives some insights about Debian technical management. Packages are the units at which software is maintained in Debian: each package is under the responsibility of a (group of) maintainer(s). A super-linear growth in the number of packages would need a super-linear growth in the number of maintainers to be sustainable in the long-term or, alternatively, an increase in the amount of packages maintained by the same people. While there is some evidence of the latter [20] on shorter time-frames (about a decade) than the one considered here (two decades), it also seems that Debian is focusing on sustainable size increases rather than trying to package every available FOSS product bearing the risk of stretching its forces too thin.

Table 6 Averages, median, and maximum of various size metrics, over packages and per release. See Table 5 for totals.

	Files		**Disk usage (KB)**		**SLOCs**		
Release	**median**	**max**	**median**	**max**	**median (K)**	**avg. (K)**	**max (M)**
hamm	65	17.8	780	0.2	4.63	25.4	1.2
slink	64	17.8	782	0.1	4.37	27.6	1.3
potato	58	27.3	732	0.2	3.46	23.2	2.0
woody	60	29.8	784	0.4	3.61	25.2	2.9
sarge	62	68.6	904	0.9	3.74	23.2	4.0
etch	65	27.2	1012	0.4	4.54	25.8	5.6
lenny	66	59.6	1000	0.9	4.41	26.5	5.9
squeeze	69	57.2	960	2.3	4.17	29.7	7.9
wheezy	69	182.4	924	2.8	3.97	36.2	13.9
jessie	67	182.4	808	2.8	3.40	37.3	14.9

Package size Thanks to the mapping between metrics and packages, we can also study the distribution of package sizes in different Debian releases: it is plotted in Fig. 3 for selected releases. Averages, medians, and maximums of selected metrics over packages are given in Table 6.

Increasingly, more and more very large packages are present in Debian: at the time of Jessie the *chromium-browser* and *linux* packages have, respectively, more than 15M and 12M SLOC. When Hamm was released its biggest package was *xfree86*, with “only” 1.2M SLOC. At the same time the per-release averages of package size are going up, whereas medians are going down. Overall it appears that: i) smaller and smaller packages are getting added to Debian, ii)

larger and larger packages are getting added too; with (ii) dominating more and more the total size of releases. A possible explanation for (i) comes from the packaging of relatively new software ecosystems that are increasingly releasing very small packages, e.g., Python's PyPi, R's CRAN, Node.js' NPM, etc. (ii) on the other hand seems due to behemoth software packages such as Web browsers, that are becoming self-contained work environments that need to (re)implement more, and more complex, functionalities that were historically available from separate packages.

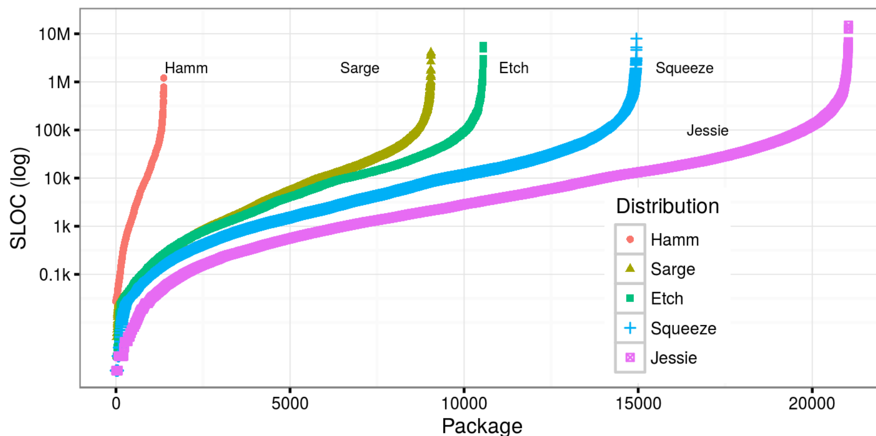


Fig. 3 Size of packages per distribution (measured in SLOC, y-axis). Each integer in the x-axis represents one package. E.g., in Jessie ≈ 7500 packages have sizes less or equal to 1k SLOC, while ≈ 20 000 packages have sizes less or equal to 100k SLOC.

5.2 Package maintenance

RQ ii is about the amount of changes that Debian users can expect when upgrading from one stable release to another. As the pairs ⟨name, version⟩ uniquely identify packages throughout Debian history, and as those pairs are available in the *package_info* table, we can leverage the Debsources Dataset to compare the sets of packages shipped by different Debian releases. Furthermore we can dissect package versions into their upstream and Debian-specific parts (see Section 3) to related changes in the Debian archive to upstream ones.

The top-half of Table 7 summarizes the amount of changes between pairs of Debian releases. *Common* packages are those that appear in both releases, in the same or different versions. *Unchanged* packages appear in both releases with the same "upstream" version, ignoring Debian-specific version changes (hence: $unchanged \subseteq common$).

It is interesting to note that 73 packages have remained at the same upstream version between Hamm and Jessie, for more than 17 years, whereas

Table 7 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**
hamm	1324c 842u	1198c 463u	1079c 270u	958c 175u	864c 148u	782c 124u	719c 100u	670c 81u	649c 73u
slink		1657c 742u	1455c 384u	1281c 252u	1155c 210u	1037c 172u	941c 136u	881c 113u	852c 101u
potato			2456c 935u	2118c 551u	1881c 436u	1683c 352u	1497c 271u	1399c 220u	1348c 201u
woody				4588c 1688u	3953c 1156u	3497c 908u	3018c 633u	2786c 520u	2648c 458u
sarge					7671c 3832u	6828c 2597u	5896c 1717u	5349c 1367u	5042c 1164u
etch						9230c 4578u	8033c 2906u	7212c 2203u	6778c 1813u
lenny							10823c 5271u	9624c 3673u	8999c 2928u
squeeze								13098c 6802u	12201c 4890u
wheezy									16160c 8427u

	from previous suite to								
	slink	**potato**	**woody**	**sarge**	**etch**	**lenny**	**squeeze**	**wheezy**	**jessie**
modified pkgs	556m	1305m	3127m	4462m	2879m	3287m	4128m	4466m	4881m
changed files per pkg	54.6%	64.4%	65.3%	67.5%	58.9%	59.8%	60.4%	57.3%	54.7%

their Debian revisions have evolved. Among these packages we can find for instance *netcat*, a network tool that hasn't changed upstream for that long, but seems to be still working just fine in Debian (otherwise it would have been removed from recent releases). This hints at the fact that long lasting unchanged packages might have been abandoned upstream, but are still maintained in Debian via patches applied by distribution maintainers.

The bottom-half of Table 7 focuses on upgrades from a release n to the immediately subsequent release $n+1$, which is the most common (and the only officially supported) upgrade path in Debian. The table shows the number of *modified* packages between consecutive releases (packages which exist in both releases, but in different upstream versions), as well as the proportion of source code files updated in these packages. The latter can be computed using the already discussed mapping between files and packages, together with either SHA1 or SHA256 checksums, both available in the *files* table.

The percentage of common and unchanged packages w.r.t. the previous release oscillates around 87% (common) and 43% (unchanged) with low variance. This suggests that Debian users experience high stability in terms of which packages are available across releases (almost 90%), as well as a steady flow (around 60%) of new upstream releases that are incorporated by Debian maintainers. The number of changed files per package on the other hand gives insights into how much new upstream releases touch the actual source code that form packages. This measure is also pretty stable across all Debian releases, ranging between 54% and 67%. Note however that this does not tell us how much individual files have been changed, only how many of them have: bumping copyright year in a file header or rewriting the file from scratch will still account for one source code file change. More precise evaluations of "how much" source code has changed can be performed leveraging TLSH hashes, that are readily available in the Debsources Dataset as well.

5.3 Programming language popularity

To address RQ iii (programming language popularity) we can simply aggregate per-package first, and per-release then, the SLOC counts available in the *metrics* table as computed by both `sloccount` and `cloc`. In either case the field *st_value* is used to detail the detected programming language. For consistency with RQ i, in the following we present `sloccount` results.

The evolution of programming languages in Debian is presented in Table 8 and plotted in Fig. 4 and 5. In both cases we restrict presented results to the most popular languages, using the Jessie release as a reference. Fig. 4 shows the evolution of language popularity in absolute SLOCs, while Fig. 5 shows the proportion over release size measured in SLOC.

Results show that C has always and still is the dominant language in Debian, since a big part of the core operating system (the Linux kernel, the GNU suite, etc) is written in C. However, while the absolute amount of C code has been steadily increasing, its proportion over the total is decreasing since

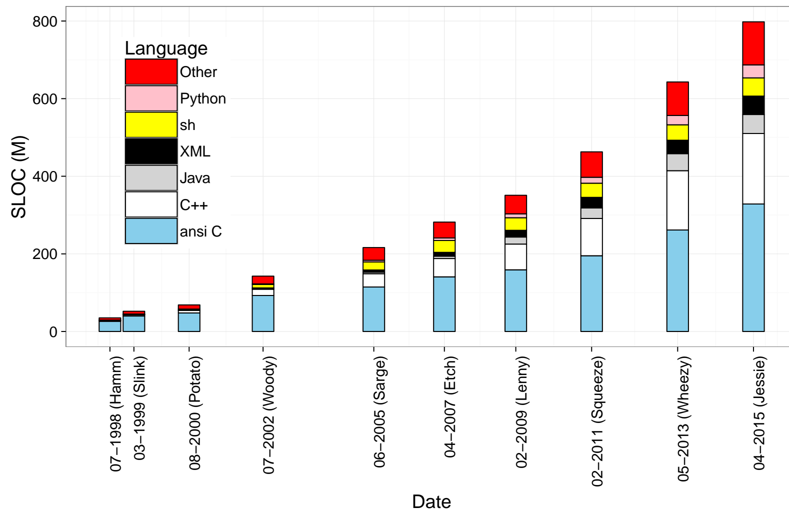


Fig. 4 Evolution of the most popular (top-6 plus other) programming languages in Debian by total number of SLOC per release.

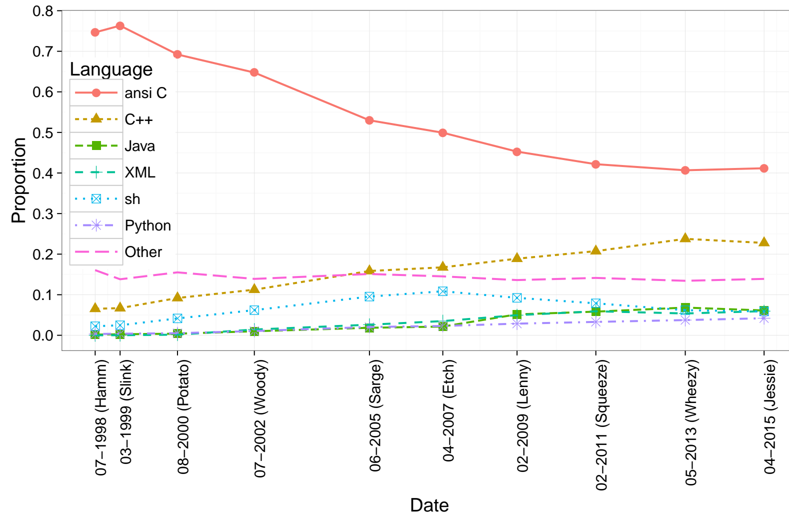


Fig. 5 Evolution of the most popular (top-6 plus other) programming languages in Debian as a proportion of release size in SLOC.

the Slink release (1999). Other languages, and most notably C++, are getting more and more relevant. The proportion of C code seems to have been stable for the past 3 releases though, at about 41% of the total.

Without a comprehensive reference base for FOSS source code it is impossible to determine how representative these numbers are of out-of-Debian trends. But the comparison with programming languages trends on other plat-

Table 8 Most popular programming languages in Debian releases, in MSloc. Numbers between parentheses represent percentage of total. (Quantities < 0.1 have been omitted and replaced by ≈ 0.)

Release	total	ada	ansic	asm	cpp	erlang
hamm	35	0.24 (0.68)	27 (77)	0.39 (1.1)	1.9 (5.5)	NA (NA)
slink	52	0.26 (0.51)	4.5 (78)	0.64 (1.2)	3.0 (5.7)	NA (NA)
potato	69	0.42 (0.61)	49 (7.6)	0.57 (0.83)	5.8 (8.5)	0.21 (0.30)
woody	15	0.58 (0.41)	94 (67)	2.6 (1.9)	15 (1.4)	≈ 0 (≈ 0)
sarge	21	1.1 (0.53)	120 (56)	2.8 (1.3)	33 (16)	≈ 0 (≈ 0)
etch	270	0.76 (0.28)	140 (53)	4.5 (1.6)	46 (17)	0.69 (0.25)
lenny	330	0.85 (0.26)	160 (49)	4.1 (1.2)	64 (19)	0.82 (0.25)
squeeze	440	1.3 (0.29)	210 (46)	4.8 (1.1)	96 (22)	1.3 (0.28)
wheezy	640	1.6 (0.25)	290 (46)	8.2 (1.3)	150 (23)	1.6 (0.25)
jessie	780	1.8 (0.23)	360 (46)	1.5 (1.3)	180 (23)	1.8 (0.23)

Release	f90	fortran	haskell	java	lisp
hamm	≈ 0 (≈ 0)	0.70 (2.0)	NA (NA)	≈ 0 (0.17)	0.11 (0.32)
slink	≈ 0 (≈ 0)	1.0 (2.0)	≈ 0 (≈ 0)	0.13 (0.25)	2.5 (4.8)
potato	≈ 0 (≈ 0)	1.4 (2.1)	≈ 0 (≈ 0)	0.27 (0.40)	3.4 (4.9)
woody	≈ 0 (≈ 0)	2.3 (1.6)	0.28 (0.20)	1.4 (1.0)	5.1 (3.7)
sarge	≈ 0 (≈ 0)	2.9 (1.4)	0.98 (0.47)	4.0 (1.9)	6.9 (3.3)
etch	≈ 0 (≈ 0)	2.1 (0.76)	0.58 (0.21)	6.1 (2.2)	7.2 (2.6)
lenny	0.29 (≈ 0)	2.3 (0.68)	0.67 (0.20)	18 (5.4)	8.1 (2.4)
squeeze	0.76 (0.17)	2.5 (0.56)	0.93 (0.21)	27 (6.1)	9.7 (2.2)
wheezy	1.1 (0.17)	8.2 (1.3)	1.6 (0.25)	44 (7.0)	8.8 (1.4)
jessie	7.5 (0.95)	9.7 (1.2)	2.0 (0.25)	50 (6.3)	11 (1.4)

Release	makefile	ml	objc	pascal	perl	python
hamm	2.3 (6.7)	≈ 0 (0.26)	≈ 0 (0.16)	0.17 (0.49)	≈ 0 (≈ 0)	0.49 (1.4)
slink	0.15 (0.28)	≈ 0 (0.11)	0.22 (0.43)	≈ 0 (0.10)	0.79 (1.5)	0.20 (0.39)
potato	0.21 (0.31)	0.15 (0.22)	0.41 (0.60)	0.31 (0.45)	1.4 (2.0)	0.36 (0.52)
woody	0.37 (0.26)	0.38 (0.27)	0.55 (0.39)	0.43 (0.31)	3.0 (2.1)	1.5 (1.1)
sarge	0.55 (0.26)	0.76 (0.36)	0.76 (0.36)	1.4 (0.65)	6.3 (3.0)	4.4 (2.1)
etch	0.68 (0.25)	1.3 (0.47)	1.0 (0.37)	1.1 (0.41)	8.1 (3.0)	6.5 (2.4)
lenny	0.75 (0.23)	1.8 (0.55)	1.1 (0.32)	0.87 (0.26)	9.4 (2.8)	1.1 (3.0)
squeeze	0.69 (0.16)	2.6 (0.59)	1.2 (0.27)	3.8 (0.84)	13 (2.9)	16 (3.5)
wheezy	0.66 (0.10)	3.8 (0.59)	1.7 (0.26)	4.4 (0.69)	18 (2.8)	25 (3.9)
jessie	0.72 (≈ 0)	4.1 (0.52)	1.9 (0.25)	5.5 (0.70)	20 (2.5)	35 (4.5)

Release	php	ruby	sh	sql	tcl	yacc
hamm	≈ 0 (≈ 0)	≈ 0 (≈ 0)	0.92 (2.6)	≈ 0 (≈ 0)	0.35 (1.0)	0.19 (0.54)
slink	≈ 0 (≈ 0)	≈ 0 (≈ 0)	1.5 (2.9)	≈ 0 (≈ 0)	0.50 (0.97)	0.25 (0.48)
potato	≈ 0 (≈ 0)	≈ 0 (≈ 0)	3.3 (4.8)	≈ 0 (≈ 0)	0.67 (0.97)	0.32 (0.47)
woody	0.58 (0.41)	≈ 0 (≈ 0)	9.5 (6.8)	≈ 0 (≈ 0)	1.2 (0.86)	0.45 (0.32)
sarge	1.8 (0.87)	0.46 (0.22)	21 (1.2)	≈ 0 (≈ 0)	2.1 (1.0)	0.56 (0.26)
etch	3.0 (1.1)	1.2 (0.45)	31 (12)	0.51 (0.19)	1.7 (0.64)	0.65 (0.24)
lenny	4.0 (1.2)	2.0 (0.61)	33 (9.9)	0.66 (0.20)	1.9 (0.58)	0.67 (0.20)
squeeze	4.7 (1.1)	4.3 (0.96)	38 (8.6)	1.5 (0.34)	2.5 (0.55)	0.81 (0.18)
wheezy	5.8 (0.92)	4.2 (0.66)	42 (6.6)	2.4 (0.38)	2.6 (0.41)	1.0 (0.16)
jessie	8.1 (1.0)	5.2 (0.66)	49 (6.2)	3.9 (0.49)	3.1 (0.40)	1.2 (0.15)

forms (e.g., GitHub [15]) is striking. Whereas GitHub developers seem to be flocking to JavaScript, Java, Ruby, and PHP, a foundational operating system

like Debian is still prominently composed of system-level languages like C and C++.

Table 9 Median file size (in SLOC) per language for the most popular languages.

Release	ada	ansic	asm	cpp	erlang	f90	fortran	haskell	java	lex	lisp
hamm	40	69	26	62	-	47	67	-	44	180	130
slink	38	68	43	60	-	11	61	12	36	190	120
potato	40	71	34	57	170	11	73	44	34	170	120
woody	47	86	75	64	42	16	81	36	37	210	140
sarge	47	79	40	66	43	38	89	37	43	180	120
etch	43	80	32	64	210	61	79	57	46	170	120
lenny	42	79	25	61	200	89	78	44	44	180	130
squeeze	45	76	20	62	160	96	76	58	45	190	120
wheezy	47	80	23	66	130	98	110	32	47	190	110
jessie	38	75	21	65	110	79	96	39	47	210	110

Release	make	ml	objc	pascal	perl	php	python	ruby	sh	sql	tcl	yacc
hamm	43	26	150	140	62	≈ 0	59	≈ 0	20	19	87	520
slink	42	22	120	12	66	≈ 0	61	≈ 0	23	16	92	510
potato	42	32	140	63	63	19	68	31	23	17	97	600
woody	46	35	170	72	61	45	65	31	27	16	80	320
sarge	47	44	160	92	63	39	60	38	35	11	94	320
etch	47	50	150	300	70	46	59	45	38	11	92	320
lenny	44	49	150	230	64	46	58	42	39	11	84	320
squeeze	32	50	140	84	56	44	59	38	37	26	69	320
wheezy	15	48	130	79	58	39	60	36	33	20	72	340
jessie	11	43	110	88	53	37	62	32	34	20	65	350

File size We can drill down to investigate median file sizes (in SLOC) per language and their evolution over time. Detecting the programming language of each source file can be done in a number of ways, each one with their own strengths and weaknesses. Two ways of doing that is delegating language detection to `cloc` and `sloccount`, using the *st_value* column of the *metrics* table; alternatively one can look at file extensions (field *ext* in table *path_info*) and map popular extensions to programming languages. We have adopted the `sloccount` approach in the following.

Table 9 presents per release and per language median file sizes for the most popular languages. For the top-6 of them, their evolution over time is plotted in Fig. 6.

Most of the studied languages are shown to be relatively stable in their median file size over time. This is the case for mainstream languages such as C, C++, and Java, as well as several others such as Perl and Lisp (not plotted). Median file size also appears to be a rather intrinsic characteristic of a programming language, that is not really affected by how popular the language is in a large FOSS ecosystem.

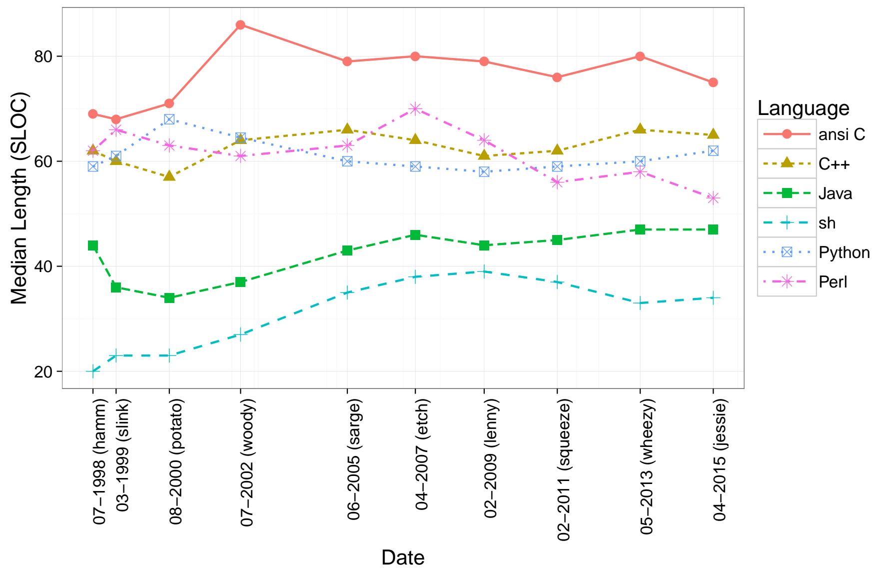


Fig. 6 Evolution over time of the median file size (in SLOC) per language, based on file extension.

5.4 Debian licensing over time

One of the most important characteristics that define a FOSS component is its license. It is very important to know the license of a package, as it determines the rights and obligations of anybody wanting to reuse and further distribute the software (either as a component or as a stand-alone product). Since the conception of Debian in 1993 the FOSS license landscape has evolved significantly. Many licenses released new versions, others have been created, and some ceased to be used. The long history of Debian creates a perfect subject to evaluate how FOSS licenses use has evolved over time, and the popularity of licenses currently in use.

Creating a census of licenses used in a large software distribution is not an easy task though. The first challenge is how to identify the licenses of individual files. Then, one needs to consider the overall licenses of aggregate/composite software bundles, such as packages in the case of FOSS distributions. The license of a package as a whole is indeed not necessarily the same of the files that compose it: due to how license compatibility works, a package might have most of its files under license A, but its aggregate license might end up being license $B \neq A$. Unfortunately there is no well-established convention for documenting either one, and tools for license identification have thus far focused on discovering the license of individual files. In order to answer RQ iv, in the following we focus on the license of individual files. We do not study the license of packages as wholes. We do, however, aggregate file licenses by package in order to study license variability within packages, answering RQ v.

Finally, in order to answer RQ vi (*how has license use evolved in Debian over time?*), we analyze the evolution of our answers to RQ iv (file licensing) and RQ v (source package licensing) across all Debian stable releases.

As automatic license identification of a file is still difficult and error prone, we avoid developing in house heuristics and rather resort to the two tools that are considered the state of the art in license identification: `ninka` [8] and `fossology` [9]. Both tools are capable of identifying commonly used licenses, but vary in the way that they deal with less common ones. For example, `ninka` is capable of identifying many variants of the BSD and MIT family of licenses, while `fossology` groups them into "MIT-style" and "BSD-style". On the other hand `fossology` is capable of identifying more licenses and, where applicable, uses the license identifiers standardized by SPDX [22] for its reports.

In terms of Debsources Dataset use, the license information we have extracted using both `ninka` and `fossology` are readily available from table *licenses*. The detected licenses of individual files as returned by the two tools can be discriminated using the *oracle* field of that table, whose value will be either "ninka" or "fossology". Note that the detected license, available in field *license*, is tool-dependent: we have favored preserving the full information returned by license identification tools over data uniformity. For the sake of brevity in the following we only discuss `fossology` results, but interested scholars can use the Debsources Dataset to explore `ninka` results.

While the Debsources Dataset contains the output of `ninka` and `fossology` for every unique file, in the following we only report about licensing of source code files (excluding, e.g., binary files such as raster images). To that end we will ignore all files not recognized as being source code by `sloccount`.

5.4.1 Individual file licensing

Table 10 Number of different licenses identified in each release.

Release	Licenses
hamm	281
slink	326
potato	437
woody	620
sarge	949
etch	1125
lenny	1352
wheezy	1879
jessie	2039

Table 10 shows the total amount of *different* licenses identified for each release in the dataset. As it can be observed, the number of identified licenses is very large and has grown an order of magnitude across Debian history. Part of the reason is that, when a file is licensed under two or more licenses, such combination of licenses is considered to be a different license by license identification tools. For example, in several Debian releases Firefox is licensed under

a combination of the MPL, GPL-2.0, and LGPL-2.1. In most releases, few licenses account for most of the identified licenses: the top 50 most frequently identified licenses (including "No_license_found") correspond to 94–97% of release source files.

Table 11 shows the top identified licenses in the oldest and newest Debian releases available in the dataset. As it can be observed, most frequently files do not have a license that `fossology` can directly identify. Fig. 7 shows the evolution over time of the most common identified licenses. As it can be seen, the most used licenses have been the GPL and BSD families, with recent increases for Apache-2.0 and the Mozilla Public License (MPL).

Table 11 Top identified licenses in two selected releases.

Release	License	Files	Prop.(%)	Accum. (%)
Hamm	No_license_found	72,533	47.5	47.5
	GPL-2.0+	22,983	15.1	62.6
	LGPL-2.0+	14,608	9.6	72.2
	BSD-style	3,667	2.4	74.6
	See-doc(OTHER)	2,490	1.6	76.2
	MIT-style	2,457	1.6	77.8
	UnclassifiedLicense	2,359	1.5	79.4
	GPL	2,329	1.5	80.9
	BSD-4-Clause-UC	2,112	1.4	82.3
	See-file	1,938	1.3	83.5
	X11	1,887	1.2	84.8
Jessie	No_license_found	1,011,088	29.7	29.7
	GPL-2.0+	432,482	12.7	42.4
	Apache-2.0	168,655	5.0	47.4
	GPL-3.0+	160,233	4.7	52.1
	GPL-2.0	148,364	4.4	56.4
	LGPL-2.1+	141,747	4.2	60.6
	BSD	115,135	3.4	64.0
	LGPL-2.0+	87,153	2.6	66.5
	BSD-3-Clause	72,634	2.1	68.7
	MIT	71,022	2.1	70.8
	EPL-1.0	66,755	2.0	72.7

5.4.2 Package licensing and variability

In order to address RQ v (source package licensing), it is not practical to simply report each and every license found in very package. Instead, we develop several metrics, each one highlighting different aspects of the licensing of source code files belonging to a given package:

– How detectable are the licenses of the package source files? For this purpose we compute the proportion of files for which a license was identified over the total number of files. `fossology` reports *No_license_found* when it does not find the license of a given file, and *UnclassifiedLicense* when it finds one it does not know. Hence we consider a file to have an *identifiable*

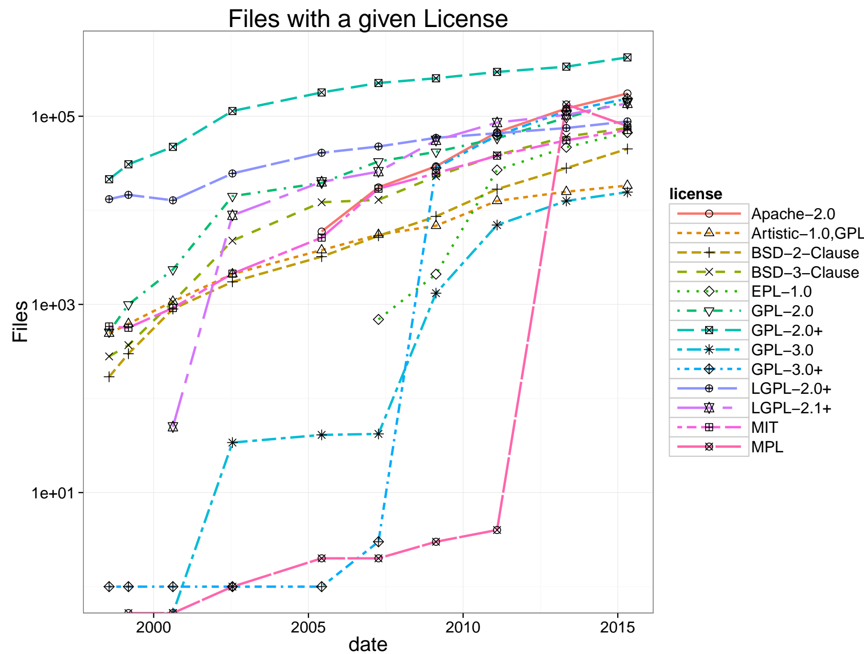


Fig. 7 Evolution of the number of files with a given license, as detected by `fossology`.

license if `fossology` reports a license other than *No_license_found* and *UnclassifiedLicense*. The *proportion of files without a license* is the number of source files without an identifiable license divided by the total number of source files.

– How many *different* licenses can be found in a given package? In this case we ignore files without an identifiable license.
– What is the *dominant license* identified in each package? We define such a license as the most commonly identifiable license, counted as the number of files it applies to. If two or more licenses are equally frequent, all of them are considered to be equally dominant.
– How much *diversity* is there in the licenses of the files of a package? The more licenses a package contains, the larger it will be the problem space of determining its license as an aggregate—due to how license compatibility works the problem will not necessarily be more *difficult*, but more options will have to be considered.
  To establish license diversity within a package we use Wilcox's Analog of the Mean Difference (MNDif). It represents the average of the absolute differences of all the possible pairs of license frequencies. Intuitively, it is the equivalent of a GINI coefficient, but applicable to categorical data. A value of 0 implies that all files have the same license, while a value of 1 that all licenses are equally represented, i.e., each license is used by the same number of files.

When aggregating by package, different licensing patterns appear. Fig. 8 shows box plots with the proportion of files that do not have a detectable license. The median number of source files without an identifiable license has fluctuated between 50% and 60%, showing the same pattern over time. It is important to mention that `sloccount` is relatively aggressive on what it considers source code. For example, `sloccount` considers Makefiles and configuration and installation scripts to be source code; these files do not normally include a license. For this reason we also include the box plots for C (Fig. 9) and Java files (Fig. 10). As it can be seen, their current median is below 5% in both cases, and over time, the proportion of files without a license keeps dropping. It seems that, at least from the point of view of `fossology` and for mainstream programming languages, FOSS development practices (and in particular writing down license annotations) are evolving in a way that makes automatic license detection easier. There is still plenty of room for improvement though.

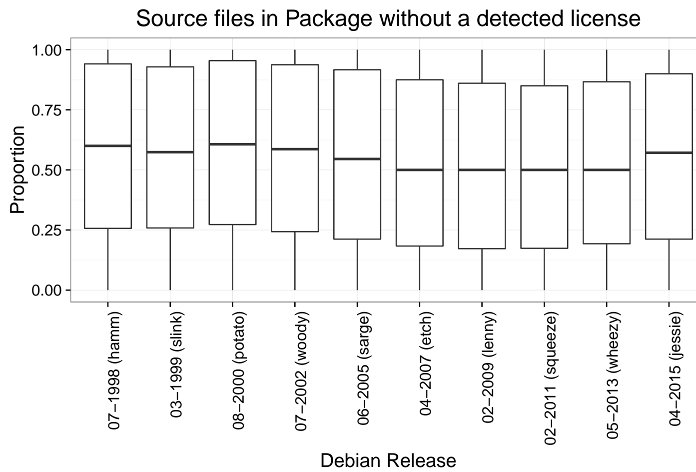


Fig. 8 Box plots of the proportion of files with no identifiable license.

The number of licenses used per package is generally very small, as shown in Fig. 11. The median is 2; the third quartile has decreased from 3 to 2 licenses in recent releases (taking into account identifiable licenses only).

With regard to the chosen licenses, we present in Fig. 12 the evolution of licenses that occur at least once in a package. As it can be seen variants of the GPL licenses are still, by far, the most commonly used, and in particular versions 2.0 and 2.0+ (i.e., "version 2 or any later version").

Fig. 13 shows the evolution of dominant licenses in Debian packages, according to our definition. The top license is, once again, GPL-2.0+, followed by: Artistic-1.0/GPL dual-licensing (the licensing choice of Perl and most Perl libraries), GPL-3.0+, and Apache-2.0.

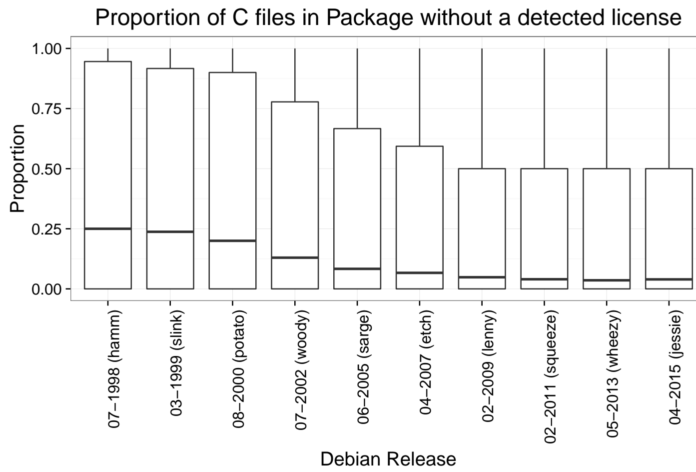


Fig. 9 Box plots of the proportion of C files with no identifiable license.

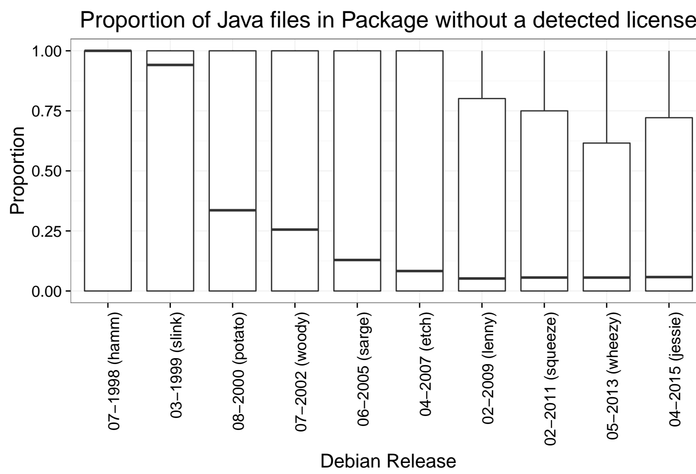


Fig. 10 Box plots of the proportion of Java files with no identifiable license.

With regard to the variability of licenses in packages, we present in Fig. 15 the box plot of the MNDif of the licenses per package in each release. As it can be seen, most packages have very small license variability, and license diversity seems to be decreasing over time. This might be due to new, popular programming language ecosystems that manage to impose, either with legal agreements or simply via “bandwagon” effects, a specific license to all the new modules and libraries that will be developed in the language.

Fig. 14 shows a scatter plot showing MNDif vs the number of source files in a package, for the most recent Debian release (Jessie). A pattern stands out: the more source files a package has, the less license diversity. This might

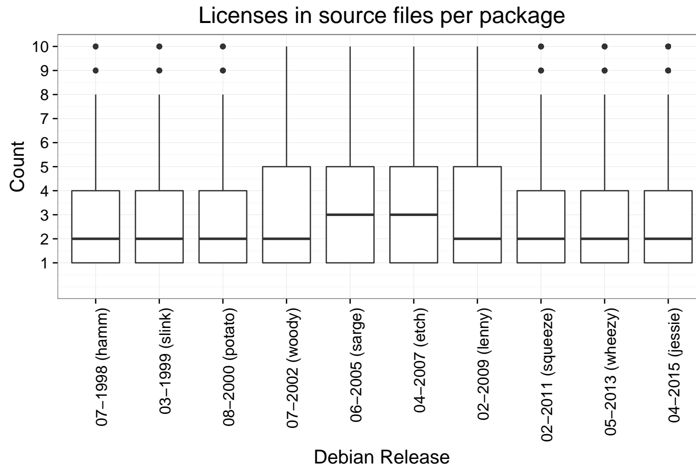


Fig. 11 Box plots with the number of different identified licenses per package.

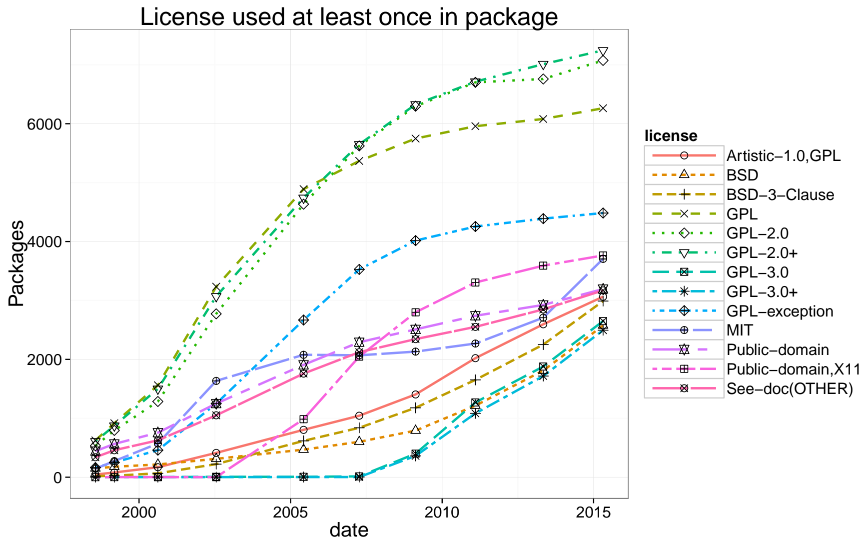


Fig. 12 Evolution of the number packages that use a license at least once.

seem counter intuitive at first, because more files would appear to give more opportunities for reusing code from other FOSS projects and hence adopt a new license, increasing diversity. Our intuition is that such aspect is countered by the fact that larger, well-established FOSS projects tend to be governance-heavy, cautious when importing external code in their own repositories (e.g., due to long-term maintainability concerns), if not simply used to impose a specific license as a condition to accept external code contributions.

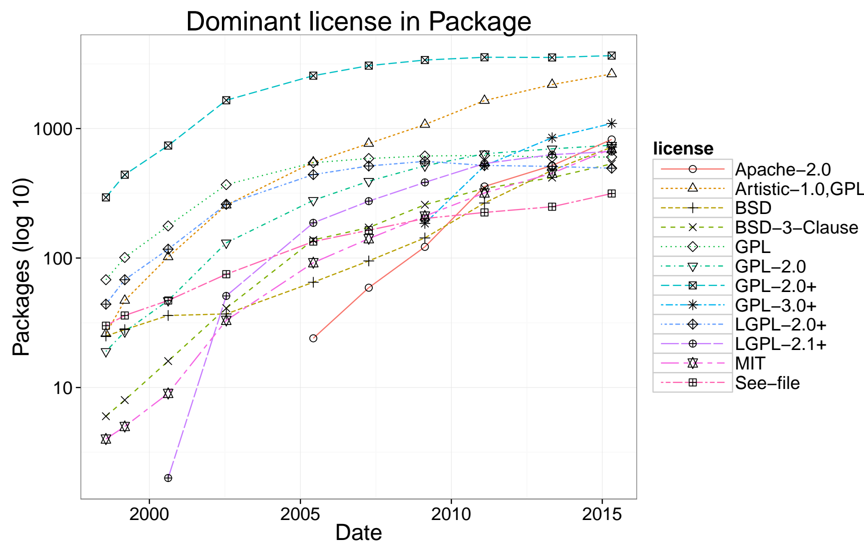


Fig. 13 Evolution of the number of packages that have a given dominant license.

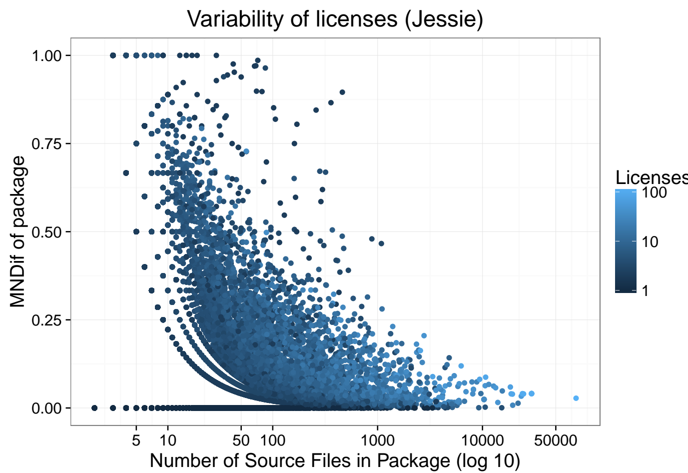


Fig. 14 Each point represents a package in Jessie (version 8): its MNDif vs number of source files. As it can be seen, smaller packages tend to have more variability in their licensing.

To the best of our knowledge this is the first study of FOSS license popularity at this scale, and in particular over such a long time frame. While there exist reports on the Web about FOSS license popularity, and most notably from Black Duck,[17] such reports do not disclose the adopted methodology nor are clear on the underlying sample of observed FOSS projects, making them non-reproducible. Furthermore they do not properly document how licenses

[17] https://www.blackducksoftware.com/top-open-source-licenses

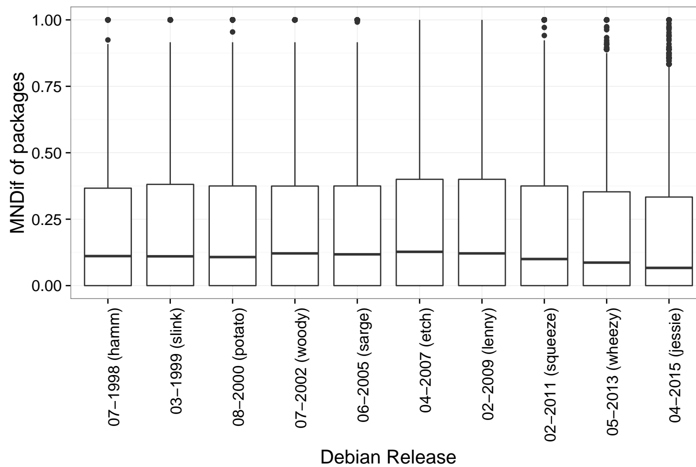


Fig. 15 Box plots showing the MNDif of packages per release. A MNDif of zero means no variability, while 1 means that every license in the package is equally represented.

are counted, which is an important and tricky aspect of surveying FOSS license use [14].

5.5 Looking back, Debsources Dataset advantages

Looking back at this case study we can attempt a self-assessment of the advantages induced by using the Debsources Dataset as a starting point. Limitations and threats to validity will be discussed in the next section.

Using Debsources Dataset metadata (the database) we have been able to study Debian growth over time as well as the correlation and distribution of the chosen metrics. We have also been able to get insights on software engineering practices such as package maintenance and study over time the popularity of programming languages and software licenses, aggregating at different granularities (file, package, release). In all cases data gathering boiled down to crafting and executing relatively straightforward (Postgre)SQL queries. Statistical analysis and plotting have then been implemented externally (using GNU R), processing Postgres query output.

Without the Debsources Dataset the starting point of the case study would have necessarily been retrieving and unpacking all Debian releases, followed by running all measurement/mining tools on the obtained source code. We have documented in Section 2.2 how, in terms of resources, doing so would have required about 1.5 months of processing time (on a single machine) and 1.5TB of disk space. None of this has been necessary to run our case study. More importantly than savings in computational resources though, the Debsources Dataset relieves scholars from the responsibility of figuring out which-tools-

to-use-when in order to mine Debian-specific data sources; the starting point becomes a relatively straightforward ER data model.

A counter argument here is that the metrics and information we were interested in were all already available in the dataset; unsurprisingly, given we initially included them in the dataset for our own needs. The first response to this is that a significant part of the metadata included in the dataset is intrinsic to either how Debian works (e.g., Debian release information) or the nature of the referenced objects (e.g., file size, SHA checksums). We expect that most studies interested in using Debian as a FOSS sample will need these information; the Debsources Dataset alleviates the need of having to mine them over and over again.

Second, when it comes to mining new facts that are not included in the Debsources Dataset, the source code part of the dataset and how it is organized offers many benefits. Most notably it saves space; thanks to deduplication the required disk space is cut by approximately 50%. As many kind batch source code analyses are I/O bound, a similar saving in processing time should generally be expected as well. Source code organization also simplifies analysis, since the files are sharded into a (relatively speaking) small number of directories; for example, one can run `cloc` recursively only 256 times, once per each top-level directory. Another benefit is that this code organization avoids the challenges of having to deal with the original path names. Some of these paths use extended character sets, or include characters that might not be handled properly by (buggy) research tools (e.g., apostrophes and white spaces), or might differ only in capitalization resulting in name clashes on some popular file systems.

It is also reasonable to expect that newly mined facts from Debian source code will need to be correlated, one way or another, with metadata that *are* available in this dataset. It is at the border of source code mining and related metadata that the Debsources Dataset offers the most time- and space-saving opportunities for empirical software engineering scholars.

6 Threats to validity

Due to the fact that at the time Debian neither had a source package format that can be extracted using today's `dpkg-source`, nor package indexes (`Sources` files), the Debsources Dataset does not include the first 3 Debian releases: Buzz (1996), Rex (1996) and Bo (1997). The first release included in the dataset is Hamm (1998). Additionally, due to a regression in `dpkg-source`,[18] 12 packages from historical releases cannot be extracted and are missing from the dataset. We do not expect such a tiny number of packages to significantly impact the usefulness of the dataset.

It is important to note that the Debsources Dataset does not fully round-trip with Debian mirrors: it is not possible to fully reconstruct source packages

[18] http://bugs.debian.org/740883

by only using the dataset. This is because the dataset is not supposed to be as precise as a backup system in capturing detailed file characteristics such as ownership, permissions, and extended attributes.

`sloccount` and Exuberant Ctags are starting to show their age and suffer from a lack of active maintenance. Most notably, they do not support languages like Scala and JavaScript, which might then be underrepresented in the dataset. The case of JavaScript is particularly worrisome, due to its increasing popularity for server-side Node.js applications. On the front of SLOC counting the issue is mitigated by the presence of counts as returned by `cloc`, which is a more modern tool with support for recent languages. At the same time we consider important to *also* have `sloccount` results in the dataset, as it is used as a reference tool in many works in the literature.

Regarding licensing data, the main threat to construct validity is the reliability of license identification as implemented by the tools that we used to detect licenses. `fossology` is a mature tool, widely used in the software industry; `ninka` is more experimental but shines in specific areas such as discriminating among license variants. With respect to external validity, we make no specific claims. While Debian is a good proxy for well-established FOSS products, Debian requires clear licensing on any software that it incorporates, hence what is observable in Debian might not reflect all of FOSS.

More generally, while we claim that the Debsources Dataset is representative, by construction, of Debian trends, any extrapolation of findings based on this dataset to more general FOSS trends should stand on its own ground. Debian is likely representative of enterprise use of FOSS as a base operating system, where stable, long-term and seldomly updated software products are desirable. Conversely Debian is unlikely representative of more dynamic FOSS environments (e.g., modern Web-development with micro libraries) where users, who are usually developers themselves, expect to receive library updates on a daily basis. Debian trends on size, language popularity, and licensing are likely not directly transferable to those contexts.

7 Related work

Debsources [3] is the software platform used to produce a previous version of the Debsources Dataset [27], which contained only metadata (no source code) and only a subset of the metadata described in this paper. Debsources can be used to recreate similar datasets for any other FOSS distribution that uses the Debian source package format, including Ubuntu[19] and hundred others [5].

Reproducing the findings of a former macro-level software evolution study [11] motivated in part the development of Debsources. That study also shows the results of running `sloccount` on Debian releases over the 1998–2007 period. The Debsources Dataset covers twice that period, offers more and more diverse metadata (ctags, disk size, checksums, license information),

[19] http://www.ubuntu.com/

and is publicly available from archival storage (Zenodo), whereas the dataset URL from [11] has been down for a few years now. Most notably the availability of the Debsources Dataset allows today to conduct studies similar to [11] without having to mirror Debian from different websites, run sloccount, manually classify by release, etc. All needed metadata are already available in an easy-to-query format. When it comes to missing metadata, they can be extracted from the source code shipped as part of the Debsources Dataset, minimizing the required computational effort thanks to source code deduplication.

The Ultimate Debian Database (UDD) [17] has assembled a large dataset about Debian and some of its derivatives, and is a popular target for mining studies and challenges [25]. UDD however lacks the time axis and is focused on distribution-level metadata. As such it lacks most of the content-oriented metadata (ctags, checksums, license information, etc.) that are available in the Debsources Dataset.

Other studies have targeted different aspects of the Debian ecosystem, such as discussion on its mailing lists (e.g., [21]). Those studies cannot be supported by the Debsources Dataset which focuses on source code. However, the presence of time-indexed metadata in the dataset allows to correlate mailing list discussions, and in particular development discussions that often touch specific versions of specific packages, with release and release dates.

Boa [6] is a Domain Specific Language (DSL) and an infrastructure to mine FOSS project collections like forges. Boa's dataset is larger in scope than the Debsources Dataset (e.g., it contains SourceForge) and also more fine grained, reaching down to the version control system level, but does not correspond to curated software collections like FOSS distributions. That has both advantages (it allows to peek into unsuccessful projects) and disadvantages: contained projects are less likely to be representative of what was popular at the time. The time horizon of BOA is also more limited than that of the Debsources Dataset.

FLOSSmole [12] is a collaborative collection of datasets obtained by mining FOSS projects. Many datasets in there are about Debian but no one is, by far, as extensive as the Debsources Dataset.

With respect to our case studies, analyses similar to the one of Section 5 on software evolution have been conducted in the past on various distributions, e.g., [24, 10], even though only punctually on individual releases. On the specific aspect of licensing, `ninka` has been used in the past to scan Debian Lenny [8]. Several empirical studies on licensing have used other FOSS distributions as data sources, such as Fedora Core 12 [7] and Debian Wheezy [26].

8 Conclusion

We have presented the Debsources Dataset: source code, release metadata about it, measurements, checksums, and license information spanning two decades of Free and Open Source Software (FOSS) history, as it can be observed through the lens of the popular Debian distribution. Using the dataset

we have conducted a long-term, macro-level evolution study of Debian looking from angles as diverse as size, package maintenance, programming language popularity, and software licensing.

The Debsources Dataset contains increasingly more fine-grained information (packages → releases → source code files → checksums → developer-defined symbols) about more than 3 billion lines of source code, from popular FOSS projects of their times. The dataset also contains the corresponding source code, deduplicated at file-level granularity, resulting in a factor 2 gain in the space required to store it in uncompressed form. Deduplication allows to efficiently process the available source code to mine further facts and correlate them with existing metadata.

The Debsources Dataset is publicly available as Open Data, documented, and reproducible using data and source code from the Debian project, as well as a variety of tools that are all available as FOSS. However, recreating it takes a non-negligible amount of storage and computational resources. Its availability as a ready to use dataset can therefore ease the work of scholars interested in macro-level software evolution, and in the history and composition of FOSS.

References

1. Pietro Abate, Jaap Boender, Roberto Di Cosmo, and Stefano Zacchiroli. Strong dependencies between software components. In *ESEM*, pages 89–99, 2009.
2. Bram Adams, Christian Bird, Foutse Khomh, and Kim Moir. 1st international workshop on release engineering (RELENG 2013). In *ICSE'13*, pages 1545–1546, 2013.
3. Matthieu Caneill and Stefano Zacchiroli. Debsources: Live and historical views on macro-level software evolution. In *ESEM 2014: 8th International Symposium on Empirical Software Engineering and Measurement*. ACM, 2014.
4. Serge Demeyer, Alessandro Murgia, Kevin Wyckmans, and Ahmed Lamkanfi. Happy birthday! a trend analysis on past msr papers. In *MSR 13: 10th Working Conference on Mining Software Repositories*, MSR'13, pages 353–362, Piscataway, NJ, USA, 2013. IEEE.
5. DistroWatch distribution search — Debian-based distributions. http://distrowatch.com/search.php?ostype=Linux&basedon=Debian&status=Active.
6. Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In *ICSE*, pages 422–431. IEEE / ACM, 2013.
7. Daniel M. German, Massimiliano Di Penta, and Julius Davis. Understanding and auditing the licensing of open source software distributions. In *18th International Conference on Program Comprehension (ICPC'2010)*, pages 84–93, May 2010.

8. Daniel M. German, Yuki Manabe, and Katsuro Inoue. A sentence-matching method for automatic license identification of source code files. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE'10, pages 437–446. ACM, 2010.
9. Robert Gobeille. The fossology project. In *MSR 2008: The 5th Working Conference on Mining Software Repositories*, pages 47–50. ACM, 2008.
10. Jesús M González-Barahona, MA Ortuno Perez, Pedro de las Heras Quirós, José Centeno González, and Vicente Matellán Olivera. Counting potatoes: the size of debian 2.2. *Upgrade Magazine*, 2(6):60–66, 2001.
11. Jesús M. González-Barahona, Gregorio Robles, Martin Michlmayr, Juan José Amor, and Daniel M. Germán. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285, 2009.
12. James Howison, Megan Conklin, and Kevin Crowston. FLOSSmole: A collaborative repository for FLOSS research data and analyses. *IJITWE*, 1(3):17–26, 2006.
13. Ian Jackson et al. Debian policy manual. Available at https://www.debian.org/doc/debian-policy/, 1996.
14. Michael Kerrisk. Surveying open source licenses. Available at https://lwn.net/Articles/547400/, 2013.
15. Alyson La. Language trends on github. Available at https://github.com/blog/2047-language-trends-on-github, 2015.
16. Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.
17. Lucas Nussbaum and Stefano Zacchiroli. The ultimate debian database: Consolidating bazaar metadata for quality assurance and data mining. In *MSR*, pages 52–61. IEEE, 2010.
18. Jonathan Oliver, Chun Cheng, and Yanggui Chen. Tlsh - a locality sensitive hash. In *CTC, 4th Cybercrime and Trustworthy Computing Workshop*, pages 7–13. IEEE, 2013.
19. Frederick P. Brooks, Jr. *The mythical man-month: essays on software engineering*. Addison-Wesley, 2 edition, 1995.
20. Gregorio Robles, Jesus M Gonzalez-Barahona, and Martin Michlmayr. Evolution of volunteer participation in libre software projects: evidence from debian. In *Proceedings of the 1st International Conference on Open Source Systems*, pages 100–107, 2005.
21. Sulayman Sowe, Ioannis Stamelos, and Lefteris Angelis. Identifying knowledge brokers that yield software engineering knowledge in oss projects. *Information and Software Technology*, 48(11):1025–1033, 2006.
22. Kate Stewart, Phil Odence, and Esteban Rockett. Software package data exchange (SPDX™) specification. *International Free and Open Source Software Law Review*, 2(2):191–196, 2011.
23. Andrew Tridgell. *Efficient algorithms for sorting and synchronization*. PhD thesis, Australian National University Canberra, 1999.

24. David A Wheeler. More than a gigabuck: Estimating GNU/Linux's size. `http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.1.03.html`, 2001.
25. Jim Whitehead and Thomas Zimmermann, editors. *Mining Software Repositories, MSR 2010*. IEEE, 2010.
26. Yuhao Wu, Yuki Manabe, Tetsuya Kanda, Daniel M. German, and Katsuro Inoue. A method to detect license inconsistencies in large-scale open source projects. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 324–333, Piscataway, NJ, USA, 2015. IEEE Press.
27. Stefano Zacchiroli. The Debsources dataset: Two decades of Debian source code metadata. In *MSR 2015: The 12th Working Conference on Mining Software Repositories*, pages 466–469. IEEE, 2015.