Noname manuscript No.
(will be inserted by the editor)

An Empirical Study of Integration Activities in Distributions of Open Source Software

Bram Adams, Ryan Kavanagh,
Ahmed E. Hassan, Daniel M. German

Accepted in Empirical Software Engineering (EMSE), Springer (January 2015)

Abstract Reuse of software components, either closed or open source, is considered to be one of the most important best practices in software engineering, since it reduces development cost and improves software quality. However, since reused components are (by definition) generic, they need to be customized and integrated into a specific system before they can be useful. Since this integration is system-specific, the integration effort is non-negligible and increases maintenance costs, especially if more than one component needs to be integrated. This paper performs an empirical study of multi-component integration in the context of three successful open source distributions (Debian, Ubuntu and FreeBSD). Such distributions integrate thousands of open source components with an operating system kernel to deliver a coherent software product to millions of users worldwide. We empirically identified seven major integration activities performed by the maintainers of these distributions, documented how these activities are being performed by the maintainers, then evaluated and refined the identified activities with input from six maintainers of the three studied distributions. The documented activities provide a common vocabulary for component integration in open source distributions and outline a roadmap for future research on software integration.

1 Introduction

Software reuse is "the use of existing software or software knowledge to construct new software" [29]. Reuse roughly consists of two major steps [4]: 1. identifying a suitable component to reuse, and 2. integrating it into the target system. For

Bram Adams
MCIS, Polytechnique Montréal, Canada
E-mail: bram.adams@polymtl.ca

Ryan Kavanagh, Ahmed E. Hassan
Queen's University, Canada
E-mail: {ryan,ahmed}@cs.queensu.ca

Daniel M. German
University of Victoria, Canada
E-mail: dmg@uvic.ca

example, vendors of mobile phones typically reuse an "upstream" (i.e., externally developed) operating system component in their device, customized with proprietary device drivers, control panels and utilities [45]. Reuse is very commonplace, as shown in studies on software projects of different sizes in China, Finland, Germany, Italy and Norway [13, 40, 41, 45, 50, 51]. For example, almost half of the Norwegian software companies reuse "Open Source" (OSS) in their products [41], while 30% of the functionality of OSS projects in general reuse existing components [77].

Although reuse speeds up development, leverages the expertise of the upstream project and, in general, improves the quality and cost of a product [4, 32, 80], it is not entirely risk- and cost-free. In particular, the integration step of reuse consumes a large amount of effort and resources [8, 11, 25, 64], for various reasons. "Glue code" [94] needs to be developed and maintained to make a component fit into the target system, and developers need to continuously assess the impact on this glue code of new versions of the component (such a new version can bring an unpredictable set of bug fixes and features). Furthermore, the component might depend on other components, whose bugs could propagate to the target system in undocumented ways [27, 57, 66, 84]. The ability to make local changes to the source code of a reused component introduces even more challenges, since an integrator typically is not familiar with the reused component's code base and hence can easily introduce bugs in such local changes [40, 52, 59, 79, 83, 90]. Worse, if the local changes are not contributed back to the owner of the reused component, the organization that made the changes will need to maintain them and possibly re-apply them themselves to future versions of the component [78, 90].

Thus far, most of the empirical studies on integration of components [11, 40, 52, 59, 64, 79, 90] concentrated on the base case of integrating one component in a target system. In practice, however, organizations tend to integrate not one, but two or more components, which brings along a set of unique challenges [54, 64, 90], especially given the popularity of open source development: in the timespan of one release, an organization needs to co-ordinate the integration of updates by multiple vendors, typically with totally independent release dates [8, 11]. For example [45], Nokia's N800 tablet platform reused 428 OSS components, 25% of which were reused as is (e.g., bzip2 and GNU Chess), 50% were changed locally (e.g., the graphics subsystem), and 25% were developed in-house using open source practices ("inner source", ISS). It is unclear for organizations like Nokia how to keep their system stable and secure amidst the integration of so many different components [40]. Furthermore, there is a clear need [8, 16, 59] for dedicated training and education of developers and organizations on integration, since in a world of open source they now need to collaborate with the providers of 3rd party components and other external contributors to benefit from external contributions and to avoid having to maintain bug fixes and other customizations oneself.

This paper aims to improve the understanding of multi-component integration by empirically studying and documenting the major integration activities performed by OSS distributions [37]. An OSS distribution basically is a "packaging organization" [59, 71], i.e., an organization that integrates upstream components into a common platform (similar to product lines [60, 68]), ironing out bugs and intellectual property issues, and providing extensive documentation and training on the integrated components. Reusing an OSS component through an established distribution provides more confidence in the quality of the component [83], and

hence many companies use OSS components as the basis for products like routers, mobile phones or storage devices [46]. Examples of established OSS distributions are Eclipse, GNOME and operating system distributions like Debian or Ubuntu.

Here, we focus on operating system distributions (henceforth called "OSS distribution"), which bundle and customize OSS operating system kernels (e.g., Linux or BSD), system utilities (e.g., compilers and file management tools) and end-user software (e.g., text processors, games and browsers) with a dependency-aware package system. There are almost 400 active OSS distributions, and each year 26 new ones are born [55]. Given the growing competition, distributions need to release new features and versions in an ever shorter time frame [44, 69, 76] to millions of desktop users and server installations. To achieve this, they rely on hundreds of volunteers to integrate the latest versions and bug fixes of the tens of thousands of integrated upstream components.

We empirically studied the major integration activities of three of the most popular and successful OSS distributions, i.e., Debian, Ubuntu and FreeBSD, using qualitative analysis on an accumulated 29 years of historical change and bug data. We document these activities and the steps used to perform them in a structured format, distilling the state-of-the-practice tools and processes followed by the actors involved in the activity, providing concrete examples, and comparing our findings to prior research and integration outside the context of OSS. Six members of the maintenance community of the analyzed distributions discussed and refined the documented activities, and provided feedback on the usefulness and completeness of the activities. Similar to the concept of design patterns [33] or reference architectures [9], the documented activities can be used by (1) organizations as a common terminology for discussing and improving integration activities for components, and (2) researchers to set up a road map for research on integration, since integration remains a largely unexplored research area [38, 40, 79].

The main contributions of this paper are:

- Identification and documentation of seven major integration activities and the processes that they follow in three major OSS distributions.
- Identification of major challenges for tool support and research for integration activities.
- Evaluation of and feedback on the identified activities and challenges by six integration maintainers and release managers of the analyzed distributions.

This paper is structured as follows. First, Section 2 discusses background and related work on software integration and OSS distributions, after which Section 3 presents the design of our qualitative analysis. Section 4 documents the seven integration activities that we identified during our analysis, followed by a discussion of the open challenges that we identified (Section 5) and the evaluation of our findings by six practitioners (Section 6). We conclude with threats to validity (Section 7) and the conclusion (Section 8) of our study.

2 Background and Related Work

This section discusses background and related work on integration and open source distributions. Table 1 summarizes key technical terms that will be used throughout the paper.

Table 1 List of common technical terms related to integration and open source distributions.

term	meaning
reuse	identification and integration of a component (e.g., class or library) into a system
OSS reuse	reuse of Open Source Software
COTS reuse	black box reuse based on Commercial Of The Shelf components
ISS reuse	reuse of Inner Source Software, i.e., OSS developed in-house
integrator	organization that integrates a third party component into its product
maintainer	individual or team doing physical integration on behalf of integrator
downstream project	synonym for "integrator"
upstream project	organization (open source project or company) whose components are being integrated by another project
upstream component	component developed by upstream project that is being reused
multi-component integration	integration of more than one upstream component
packaging organization	integrator whose business goal is to package upstream components into a coherent platform that is offered for sale or reuse
package	upstream component that has been integrated into an OSS distribution using the distribution's packaging format (e.g., "rpm")
binary distribution	distribution providing compiled code in its packages
source-based distribution	distribution providing source code in its packages, for compilation on the end-user's machine
derived distribution	"child" distribution that customizes packages of an existing "parent" distribution and adds additional packages to it

2.1 Software Integration

Reuse can be black box or white box [28]. Black box reuse refers to "Commercial Off The Shelf" (COTS) components [8], for which source code typically is not available. Hence, such components can only be configured and plugged into a target system. White box reuse provides access to the component's source code to customize it to the needs of the target system, either because the component is OSS [78] or because it is developed in-house following open source principles ("inner source", ISS), a practice that is increasingly more common in large companies like Alcatel-Lucent, HP, Nokia, Philips and SAP [79]. OSS and ISS reuse are also very common in the base platform of software product lines [53, 54, 68], since up to 95% of such a platform consists of "commoditized" features readily available from upstream projects.

In general, software reuse creates a win-win situation for the reusing organization and the upstream project whose software is reused. The former benefits from the features provided by the component in terms of productivity and product quality [29,80], while the upstream project benefits financially (through licensing) and/or qualitatively from the various forms of feedback in the form of defect reports, code contributions and user experiences. However, despite the differences between COTS and OSS/ISS, all forms of reuse introduce a dependency on an upstream project (COTS/OSS) [26, 40, 49, 62, 64] or another division inside the organization (ISS) [54], which can lead to hidden maintenance costs.

Software reuse has been studied extensively from the perspective of how to make a software system reusable [15, 23, 29, 56, 67, 68], how to select components for reuse [6, 13, 50], how to resolve legal issues regarding software reuse [36], and what factors can impact collaboration between the component provider and integrators [10, 17, 42, 43, 74]. In particular, Curtis et al. found, based on interviews,

how the need to communicate outside the team, department or even company boundaries opens a can of worms (e.g., finger-pointing, silos of domain knowledge, limited communication channels, lack of contact persons and misunderstanding due to different context) that can negatively impact the integration process. Herbsleb et al. [42, 43] empirically proved that the need to involve more people indeed relates to the time necessary to resolve bugs and integration issues.

In contrast, the concrete activities involved with the integration of reused components, as well as their costs, have been studied in substantially less detail. Especially for multi-component integration, where not one but a potentially large number of (typically open source) components are being reused by an organization at the same time, empirical evidence is currently lacking [54, 64, 90]. Lewis et al. [49] note that "The greater the number of components, the greater the number of version releases, each potentially coming out at different times." Hence, what kind of activities does such integration imply, and how do those activities relate to known activities for single-component integration? Before explaining how this study addresses these questions, we first discuss prior work on COTS, OSS and ISS reuse.

COTS Reuse and Integration

Brownsword et al. [11] studied over 30 medium-to-large commercial projects to analyze the hidden integration activities of COTS reuse. They found that for an organization it is important to be informed about (new versions of) promising COTS components and to continuously monitor the impact of the components on the organization's code base. They also point out the maintenance issues of glue code and configuration of a COTS component, and the fact that projects do not control the upstream project. However, the findings are rather high-level, and do not explain *how* the projects coped with multi-component integration.

Lewis et al. [49] relate on their experience with COTS reuse in 16 government organizations. They especially stress the loss of control as soon as a contract for COTS reuse is signed: any clause or adaptation that was not negotiated will result in additional costs down the line. Changing one's own system or looking for another COTS component is preferable to requesting (and having to pay) the component vendor to adapt her component. The main question in the studied organizations' mind was "How do we upgrade an operational system without a great deal of disruption?". There was no consensus whether one should always update to the latest version of a reused component, wait until a new major version or incorporate only the most pressing changes (e.g., security fixes). These questions only aggravated for those organizations that were reusing dozens of components, which causes additional coordination issues.

A similar study was performed by Morisio et al. [64] at NASA. Again, integration was the most costly aspect of COTS reuse, yet the integration activities varied widely across projects. Glue code was the main means of integration, and the authors note that most successful projects had to stay in contact with the COTS component provider throughout the lifecycle of the system to avoid surprises in the next version of the COTS.

OSS Reuse and Integration

Merilinna et al. [59] performed a literature survey and structured interviews with nine small-to-medium Finnish companies that reuse OSS components. They found

that integration problems are primarily due to the heterogeneous environments that components need to support as well as the lack of documentation, forcing companies to rely primarily on their own experience. Merilinna et al. identified three ways to deal with integration problems: using OSS components as a COTS component (no changes to the code), contributing changes back upstream, or using a packaging organization like an OSS distribution as mediator. Not upgrading to a new version of a reused component can also help. In any case, a thorough analysis of the OSS component to be reused can avoid many problems.

Ven et al. [90] performed interviews with members of a commercial project reusing OSS components, and examined in detail the trade-off between changing the code and contributing the changes back. Even though a project wants to avoid maintaining local changes (since this is costly), the alternative of contributing changes to the upstream project also requires an investment of time and resources, for example to get to know the contribution procedures and to keep track of the future evolution of the upstream project. Even if a patch is accepted by the upstream project, the organization developing the patch might still be required to maintain it since only it has all the insight. Ven et al. recommend to contribute patches if the local changes are sufficiently generic, to maintain patches oneself if they are too specific, or (in the worst case) to fork the upstream project, even though such a fork has only a small chance of success.

While Merilinna et al. and Ven et al. identified two integration activities that we also identified in our study (i.e., `Upstream Sync` and `Local Patch`), we approached those activities from the perspective of a packaging organization (and multi-component integration) and documented them in a structured way.

ISS Reuse and Integration

Stol et al. [79] studied the emerging practice of developing and reusing code in-house using open source practices (ISS). ISS is a popular phenomenon in large companies, since it provides the benefits of OSS reuse without giving up control. Some companies only offer their employees the infrastructure for OSS reuse, while others make it part of their development strategy. A systematic literature study and detailed study of ISS inside an organization shows that the most costly ISS issues are due to integration. In addition to the integration issues related to OSS reuse in general, other challenges like backwards compatibility and the peculiar interplay between the ISS team and other teams in a company were identified. For example, the ISS team can send a "delivery advocate" to other teams to help them integrate the ISS components. However, various activities are company- and ISS reuse-specific. For example, the ISS team receives components initially from a specific team in the organization, but after integration becomes responsible for it itself and starts acting as upstream for the other teams in the organization (even though the original developers still collaborate on the development of the component). In this paper, OSS distributions and upstream projects are separate, independent entities.

Finally, van der Linden [54] reports on adoption of OSS and ISS reuse in software product lines [60,68]. The platform on which such product lines are built largely consists of common functionality for which many components are available. Reuse of OSS and ISS components for such functionality improves the quality and speed of development, however it also introduces a dependency on the upstream projects, not only from the platform, but from all products based on the platform.

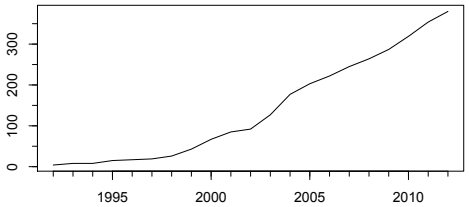

Fig. 1 The number of active Linux distributions over time. BSD distributions (e.g., FreeBSD) are *not* included.

In addition to the best practices mentioned before, close collaboration with the upstream projects in a symbiotic fashion is key to keeping track of new features and changes, and can be established by reporting or fixing bugs. Although OSS distributions can be seen as a product line, our study focuses especially on the identification and structured documentation of major integration activities in the context of multi-component integration.

2.2 Open Source Distributions

This paper focuses on the maintenance activities involved in software integration in the context of OSS distributions, since this context enables us to study integration in a multi-component, open source setting. OSS distributions are one of the most well-known open source packaging organizations [37, 71]. Such distributions integrate a collection of upstream software components consisting of an operating system kernel (e.g., Linux or BSD), core libraries, compilation tools and software for users like desktop applications and web browsers. Thanks to their inclusion in an OSS distribution, the integrated upstream projects can reach millions of users without having to market themselves. Although distributions are especially known in the Linux and BSD world, even commercial products like Microsoft Windows and Mac OS X can be considered as distributions (they just ship with more ISS projects than OSS).

There are hundreds of OSS distributions, most of which integrate thousands of upstream components. Figure 1 shows that the total number of currently active Linux distributions has grown to 380 (in addition to 135 discontinued distributions, which are not shown), increasing more or less by 26 distributions each year [55]. For the BSD family of open source kernels, there are twelve currently active distributions [92], in addition to 22 distributions that are either discontinued or have an unclear status. The most popular Linux distributions like Debian and Ubuntu both integrate more than 24,000 OSS components, whereas FreeBSD (most popular BSD distribution) integrates almost 23,000 components. The Debian distribution doubles in size every 2 years, having passed the mark of 300 MLOC in 2007 [37].

Despite this large scale, integrating an OSS project's components into a distribution goes far beyond black-box reuse. First, the upstream components need to be turned into a distributable "package". Distributions such as Debian, Ubuntu and Fedora, compile the components for a particular architecture, then split up the compiled libraries and executables across one or more "binary" packages. Such

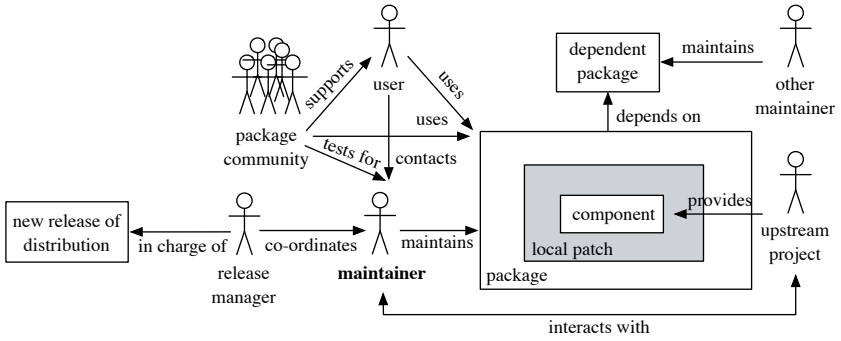

Fig. 2 The maintainer and her relation to other major stakeholders in an OSS distribution.

packages (together with the packages they depend on) can be automatically installed using a distribution-specific package management system, such as "apt", "dpkg" or "yum". Source-based distributions, like FreeBSD, distribute the (possibly customized) source code of an upstream component to the end-user as a so-called "source" package (FreeBSD uses the term "port" for this), for compilation on the user's machine. Unless otherwise specified, the term "package" in this paper will refer to both "binary" and "source" (port) packages.

After building and packaging the upstream component, the new package needs to be tested and delivered to the end-user. Once a package becomes available to end-users (including the integrators), the real integration maintenance work starts, since packages (and their dependent packages) need to be continuously updated to new versions of the packaged component. Similarly, bugs in the package should be detected and fixed promptly, and (if appropriate) patches should be sent back to the upstream project that developed the packaged component. Local changes to the package that have not been sent back, however, need to be maintained and kept up-to-date by the distribution. User complaints should be triaged and processed by the distribution as well, before escalating them to upstream, if appropriate.

Organizations that reuse a component typically [46, 59] appoint a person or group of people, i.e., the "maintainer(s)", to perform and co-ordinate integration activities on the organization's behalf. Organizations like OSS distributions dealing with multiple upstream projects and components typically have multiple maintainers, each one responsible for a group of related upstream components. Figure 2 shows the interactions of a distribution's maintainer (in bold) with the other major actors of the distribution. The maintainer packages and customizes the upstream software component by herself, interacting with the upstream project whenever necessary, for example to understand changes in a new release or to communicate reported bugs. Customizations result in local patches applied to the vanilla upstream component, after which the patched component is packaged using the distribution's package management tool. The package is being tested by the project's package community, which consists of volunteering contributors and testers. Once stabilized, packages can also be used by end-users, who can contribute bug reports or suggestions by contacting the maintainer. The maintainer's work ultimately ends up in an official release of the distribution, hence all maintainers are being co-ordinated by the release manager in charge. Some of the common activities of

the release manager is discussing release-critical bugs or project-wide packaging policies with the maintainer, and enforcing deadlines.

Given the size of a distribution, most of the maintainers are responsible for multiple components (each of which is packaged into one or more packages). Debian has around 2,400 [21] maintainers for 24,000 integrated components (a ratio of 10 components per maintainer), while FreeBSD has around 400 [31] maintainers for 23,000 components (ratio of 57.5). Ubuntu only has around 150 [86–88] maintainers for 24,000 components (ratio of 160), since most of its packages are inherited as-is from Debian, thus requiring less work. Given the high maintainer-to-component ratios, maintainers often team up to share package responsibilities, but even then, they still need to divide their attention and limited time across many components. In addition, the maintainers are not the developers of the packages that they are maintaining, which means that even more time is spent to fully understand changes or to contact the upstream developers about a change [11, 79]. Finally, various proposals have been launched to shorten the time frame in between releases of distributions [44, 69] or even to synchronize releases with those of other distributions [76]. This further complicates the task of the package maintainers.

This paper identifies and documents the integration activities that must be done on a daily basis by the maintainers of three of the most successful OSS distributions. Previous research has focused exclusively on the other stakeholders in Figure 2: the governance processes of distributions [72], release management [61, 89], the package/developer community [73], the (evolution of the) size and complexity of packages [37], and the dependencies of packages [34]. Given the central role of package maintainers in the success of a distribution, their responsibilities and challenges need to be understood in order to streamline the interaction between the OSS distribution and the upstream project, and to bring new maintainers quickly up-to-speed. Furthermore, previous work focused especially on integration of individual components, while packaging organizations like OSS distributions need to deal with the integration of thousands of components at the same time, with their users expecting the latest versions of each component to be integrated. Finally, open source development forces organizations to collaborate with external parties to reap the full benefits of quality and innovation that can be achieved with open source components. If not, organizations waste substantial effort, for example to maintain their own local patches. Hence, studying the integration activities of distributions will help us understand integration in a multi-component, open source context.

The following section presents the approach that we followed to identify and analyze the major integration activities in three large OSS distributions.

3 Case Study Setup

The goal of this paper is to empirically identify and document the major integration activities in use by packaging organizations for multi-component OSS integration, as existing empirical work focused exclusively on single-component integration. Since a wide range of packaging organizations exists, as a first step we focus on some of the most experienced integration experts in the area of OSS reuse, i.e., OSS distributions. In particular, we perform qualitative analysis on

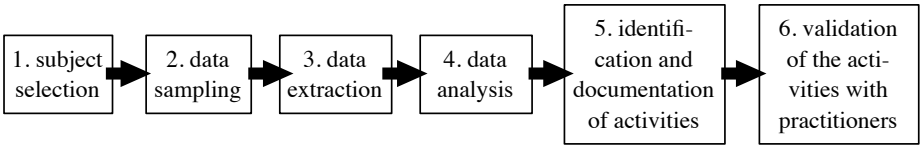

Fig. 3 Overview of our case study methodology.

three of the largest and most successful OSS operating system distributions, i.e., Debian, Ubuntu and FreeBSD.

Although our results consist of integration activities performed in OSS distributions, these activities are not unique to OSS integration, nor are they just a subset of the integration activities performed by commercial organizations. Whereas in a commercial setting organizations used to buy or develop all dependencies themselves, an OSS setting requires one to collaborate with a variety of external stakeholders to avoid being stuck with one's own patches and customizations. Avoiding this requires a different set of integration activities than before. In fact, those activities now need to trickle back into the commercial organizations that started to adopt OSS practices internally (ISS reuse).

To help such organizations, as well as open source projecs, this paper addresses the following question: What is the core set of activities in OSS for dealing with integration of multiple 3rd party components? This question allows us to empirically study what is being done in OSS integration, how it is being done and what challenges expert integrators still face. In particular, it also helps us understand what are the state-of-the-art techniques in use by OSS projects to facilitate their integration activities.

This section discusses the methodology for our study, which is also illustrated in Figure 3. We first performed a qualitative analysis to identify and document major integration activities, then evaluated these findings with stakeholders from the three distributions.

3.1 Subject Selection

To obtain a representative sample, we selected a mixture of binary and source-based, and derived and independent OSS distributions. A derived (or "child") distribution automatically inherits the packages of its "parent" distribution. It then customizes some of those packages, and also adds its own packages, in order to enforce a uniform look-and-feel, focus on specific types of packages or to specialize to a certain set of users (e.g., office workers vs. music producers). Although a derived distribution saves substantial integration time, it also leads to a unique set of integration activities, since each level of derivation adds an additional layer in the integration process.

When looking at the history of open source distributions [55], Debian and Ubuntu clearly stand out as two of the most influential distributions, with 41.0% of all distributions deriving from Debian (211 out of 380 active and 135 discontinued distributions), 90 from Ubuntu and 17 from FreeBSD. In particular, the Debian distribution has 81 child distributions, 105 distributions deriving from those child distributions ("grand-children"), 24 great-grand-children and 1 great-great-grand-

Table 2 Characteristics of the data for the three subject distributions.

	Debian	Ubuntu	FreeBSD
start of project	16/08/1993	20/10/2004	11/1993
start of data	12/03/2005	20/12/2005	21/08/1994
end of data	16/08/2011	14/09/2011	01/09/2011
#components	24,263	25,345	22,733
#packages	92,277	66,595	22,733
#pkg. versions	896,757	446,324	162,135
#releases	4	14	8 major/55 minor
#maintainers	2,400	150	400

child [55]. The latter potentially needs to integrate packages from its four ancestors as well as from some upstream OSS projects directly. Ubuntu itself has 79 children and 11 grand-children [55], while FreeBSD has 15 children, 1 grand-child and 1 great-grand-child [92].

We found that the impact of the above distributions on other distributions also translated well to their popularity in terms of number of users. In contrast to mobile app stores, there is no official popularity poll or ranking of OSS distributions. However, since May 2001 one of the leading sources on OSS distributions is the `distrowatch.com` web site, which contains announcements of new versions of distributions as well as detailed historical overviews of each distribution (either Linux- or BSD-based). One of its major features is that, on a weekly basis, the site keeps track of how many people search or click for each distribution. Although this ranking does not map 1-to-1 to the number of downloads, it does give an important indication about the popularity of OSS distributions.

Despite its age (the first Debian release was made on the 16th of August, 1993), Debian was still the fourth most popular binary distribution at the time of our case study, while Ubuntu was the second most popular binary and derived distribution. We decided not to study the top binary distribution at the time of our case study (i.e., Linux Mint), since it was a rather recent distribution derived from Ubuntu, without sufficient historical data available. The third most popular distribution was Fedora, but since this is totally unrelated to the Debian/Ubuntu ecosystem, we also did not study this distribution. As source code-based distribution, we picked the most popular source code-based BSD distribution, i.e., the FreeBSD distribution. Note that FreeBSD is also the most popular BSD distribution according to the 2005 BSD Usage Survey [39].

3.2 Data Sampling

We study integration activities by systematically analyzing, categorizing and revising historical package data for Debian, Ubuntu and FreeBSD to create a classification of integration activities. Given the large number of packages and package-versions in the three distributions (Table 2), we could not examine all of them manually. Instead, for each distribution we sampled enough package-versions to obtain a confidence interval of length 5% within a 95% confidence level, taking into account the large population size [14]:

$$sample\ size = \frac{ss}{1 + \frac{ss-1}{\#pkg.\ versions}}$$

```
clisp (1:2.49-1) experimental; urgency=low

  * Use libsigsegv 2.8 should fix a few issues. (Closes: 566686)
  * conflict against libreadline6 as it would be illegal to link
    against that. (Closes: 553741)
  * Use bash for clisp-link (Closes: #530054)
  * New upstream version (Closes: #462742)3
  * Redid debianization from scratch.
    (Closes: #504514,#177057,#462085, #488042, #462088, #433592, #433596)
  * Dropped conflicts with ancient versions of clisp.
  * Changed build system on powerpc, the resulting image
    passes all tests, so should Closes: #592768
  * We got a fix the 'file with wrong *pathname-encoding* in ~/' bug.
    (Closes: #443520)
  * Fix the build on Sparc by disabling FFI and dynamic modules completely.

 -- Peter Van Eynde <pvaneynd@debian.org>  Tue, 28 Sep 2010 07:31:59 +0200
```

Fig. 4 Example change log message for clisp package-version 1:2.49-1 in Debian.

with

$$ss = \frac{Z^2.p.(1-p)}{0.05^2}$$
$$Z = 1.96 \; for \; 95\% \; conf. \; level$$
$$p = 0.5 \; for \; pop. \; with \; unknown \; variability$$

This means that if we find an integration activity to hold for $n\%$ of the sampled package-versions, we can say with a 95% certainty that $n \pm 5\%$ of all package-versions exhibit that activity. For example, $7 \pm 5\%$ would mean that the activity would hold with a 95% certainty for 2% to 12% of the package-versions. Although the three distributions have a different number of package-versions, the asymptotic nature of the sample size formula obtained the same number of package-versions (384) for each distribution.

3.3 Data Extraction

We randomly sampled 384 package-versions from each distribution, then automatically extracted for each selected package-version the corresponding change log message. Such a change log basically consists of a detailed [46] bullet list containing a high-level, textual summary of all major changes in a particular package-version, as well as the explicit IDs of all fixed bugs. Figure 4 shows an example change log message of a Debian package-version (Ubuntu and FreeBSD use a similar format). Except for two changes, all changes in Figure 4 fix open bug reports, with the reports' identifier pasted inside the change log. As distributions stipulate that each new package-version has to be documented in a change log [22], we used change log data as starting point for the analysis of each package-version.

To interpret the change log's reported changes, we then manually analyzed the referenced bug reports via the distributions' bug repository. As explained below, each distribution uses a different technology for its change logs and bug repository, but we were able to write scripts to automate the fetching of both the logs

and reports. The bug reports often contained references to emails on a distribution's mailing lists, and sometimes contained patches that had been proposed as a possible bug fix. If present, we also studied these messages and patches. Finally, to clarify technical terms or understand particularly unclear bugs or changes, we used the distribution's developer documentation (accessible from a distribution's web site) and, in the worst case, any relevant web search, especially for finding relevant communication on online fora. This was only necessary in a small number of cases.

We now discuss how we obtained the above data for each of the three distributions. This data can be found online in the paper's replication package [2]. For Debian, we obtained the names of all integrated components across Debian's entire history from the so-called snapshot archive. This is a server containing all versions of all packages over time[1], and allowing scriptable access via a public JSON-based API. Then, for every integrated component, we retrieved all version numbers, their timestamps and the list of binary package names associated with the component (since a component can be split across multiple packages). After sampling 384 package-versions, we downloaded the corresponding change log using a simple script from Debian's change log repository[2]. Bug reports mentioned in the change logs can be found in the bug repository using the bug identifier[3]. Related email messages and other data mentioned in the bug reports was found by using a web search.

For Ubuntu, we used the Python API of the Launchpad collaboration platform[4] to retrieve the names and version numbers of all Ubuntu packages that have ever existed. Because Ubuntu is derived from Debian, we filtered the Ubuntu packages to include only the ones customized by Ubuntu, since the other packages are identical to Debian packages. Ubuntu-customized packages have a version number ending in "-MubuntuN", where "M" and "N" are numbers following a special convention. We found 133,311 of such package versions, belonging to 26,858 packages. Except for a different location of the change logs[5] and bug reports[6], we used the same approach for data extraction as for Debian.

For FreeBSD, data extraction was a bit more involved, since it is a source-based repository. For this reason, we retrieved a copy of the FreeBSD version control system (CVS)[7], which contains all local file changes ever made to all reused components. Since such CVS changes are too fine-grained to be considered a "version", but releases are too coarse-grained (multiple port versions can exist in between two official releases), we had to reconstruct the port versions by grouping related CVS changes together. For this, we used the FreeBSD convention that each port's Makefile is expected to have a PORTREVISION variable that is changed "each time a change is made to the port which significantly affects the content or structure of the derived package" [30]. If a maintainer does not change the PORTREVISION (nor the related PORTVERSION variable), the correspond-

[1] `http://snapshot.debian.org/`

[2] `http://packages.debian.org/changelogs/pool/main`

[3] `http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=XYZ` with XYZ the bug identifier

[4] `http://api.launchpad.net/1.0/`

[5] `http://changelogs.ubuntu.com/changelogs/pool/main`

[6] Manual search using the bug identifier on `https://bugs.launchpad.net/ubuntu`

[7] `ftp3.ie.FreeBSD.org::FreeBSD/development/FreeBSD-CVS/ports/`

ing changes are not deemed important enough to be automatically picked up by users during an update of their installation. We interpret this as "changes that do not change the PORTREVISION variable do not define a new port version", similar to the definition of "version" of binary packages.

In practice, we determined for each port the timestamps of all changes that change PORTREVISION and/or PORTVERSION, then grouped all changes to a port's files between two consecutive PORTREVISION changes (excluding the first PORTREVISION change) into one port version. We treated all changes up to and including the first Makefile revision as the first PORTREVISION, to account for the initial import of a port. We wrote scripts that queried the CVS repository[8] for all commit log messages between the start and end date of a port version. The change logs of the resulting port versions then correspond to the concatenation of these commit log messages. Finally, bug reports were obtained from FreeBSD's bug repository based on the bug identifiers mentioned in the change logs[9].

3.4 Data Analysis

Since we did not have any classification of integration activities to start from, initially the first author studied the Debian distribution as a pilot project. He manually interpreted the changes documented in the change log of each sampled package-version, then looked up the bug reports referenced by the change log in order to understand which bugs had been resolved or which features had been added, and how this was done. For the latter, the bug reports' comments were an important source of information. To fully understand the scope and context of more complex changes, he sometimes had to consult email messages referenced by the bug reports and patches attached to them. In case of doubt or usage of unfamiliar technical terms or inside stories, the distribution's developer documentation was considered or, in the worst case, a web search was performed.

Once it was clear what exactly the integrators had done to produce the analyzed package-version, the package-version was tagged with any observed activity to summarize the rationale behind the version. Two examples of activities could be "new release" or "package dependency change". More than one tag could be assigned to a version, since a new version of a package typically consists of multiple changes (as seen earlier in Figure 4). By repeating this procedure for all sampled Debian versions, and constantly revising already analyzed versions when new tags were found, an initial tagging schema was built up, representing different activities that go into a package-version.

After finishing the pilot project on Debian, the first two authors revised the obtained tagging schema, leveraging the second author's experience as a Debian/Kubuntu maintainer and developer. Some tags were merged, others were renamed, and with the resulting tagging schema in hand, we revised the Debian analysis to standardize the tags used. Afterwards, both authors analyzed the Ubuntu and FreeBSD data using the same tagging schema as a starting point (and using the same approach as for Debian). Conflicts in tagging between both authors were manually resolved through discussion. We did not find additional

[8] `:pserver:anoncvs@anoncvs.tw.FreeBSD.org:/home/ncvs`

[9] `http://www.FreeBSD.org/cgi/query-pr.cgi?pr=XYZ` with XYZ the bug identifier

tags for Ubuntu and FreeBSD, giving us confidence about the completeness of our initial tagging schema. Eventually, we obtained seven very popular tags, two less popular ones and a catch-all tag for multiple unique or less frequent activities unrelated to any of the other tags. We excluded the latter three tags from our analysis, but we come back to them in Section 6. The replication package [2] contains the tags and noteworthy observations of the sampled package versions.

3.5 Identification and Documentation of Activities

The seven most popular tags obtained after the manual analysis all correspond to unique integration activities, however each distribution could have its own terminology and workflow for such an activity. Hence, in order to abstract up the commonalities and variabilities across distributions for a particular activity (tag), all authors together distilled the intent, motivation, common tasks and current practices across the distributions based on (1) the information that we encountered in the change logs, bug reports and mailing lists for the sampled package-versions, as well as (2) the second author's experience as a Debian/Kubuntu developer. This was an iterative process, trying to separate the essential steps used during an integration activity from implementation details or exceptions in a particular distribution. Typically, each author would refine one or two patterns, then send to the next author for further refinement until no more changes were made to an activity.

Similar to design patterns [33], we then "captured [the activities] in a form that people can use effectively". For each integration activity, we documented in a rigid format its intent, motivation, the major tasks involved in the activity, its participants, possible interactions with other activities and notable instances of the activity in the three studied distributions (Debian, Ubuntu and FreeBSD). Interactions are based on co-occurrence of activities in our data. We also tried to compare each activity to prior work in the integration literature, to put each activity in context.

During the tagging of integration activities, and the abstraction into pattern form, the authors encountered recurring issues and problems of the package maintainers. Such issues and problems were noted down by each author individually, then compared and clustered to obtain a set of challenges, across 4 research areas. After filtering out challenges that were already addressed by related work, we obtained 13 concrete challenges or limitations that, based on our data, seemed to hold back maintainers in their activities. To cross-check those challenges, together with the activities that we documented, we performed a validation with practitioners in the next step.

3.6 Validation of the Activities by Practitioners

In order to get feedback on the correctness and usefulness of the documented integration activities and challenges, we contacted members of the package maintenance community of Debian, Ubuntu and FreeBSD. We asked them to (1) verify the correctness of the activities that we derived and abstracted from the change log, bug report and other historical data, as well as of the challenges that we

uncovered, and to (2) provide feedback on the usefulness of the activities as well as the activities and challenges that we might have missed while analyzing the sampled package-versions.

Based on their extensive experience with the 3 distribution communities, the second and fourth author first compiled a short-list of package maintainers and release engineers experienced with maintaining large packages. We then contacted the people on the short-list by email, since email is the preferred channel of communication for maintainers (and maintainers are volunteers spread across the world, without a fixed office). We played with the idea of creating a bug report for our study, since maintainers track the bug repository of their package from close-by, however since bug reports are a public broadcast medium, and people would have been able to chime in and perhaps influence the maintainer, we discarded the bug repository for our purposes.

We eventually received feedback from 3 maintainers (M1, M2 and M3) active in both Debian and Ubuntu, one (M6) in Debian, 1 (M5) in Ubuntu, and 1 (M4) in FreeBSD. All of them have at least five to ten years of experience, since the role of package maintainer or release engineer can only be deserved through years of active involvement in a distribution. Note that to respect their anonymity we will refer to all of them as "maintainers" and use a symbolic name.

When contacting the maintainers, we provided them a draft of this paper, then asked them for feedback about the documented activities and challenges. In particular, we asked the following questions to evaluate the usefulness and completeness of the activities and challenges:

Q1 What activities did we miss?
Q2 What can the documented activities be used for?
Q3 Which existing tools and techniques for these activities did we miss?
Q4 What challenges did we miss?
Q5 What promising tools/techniques do you see coming up to address some of the challenges?

The maintainers replied to the five questions by email. All six also provided higher-level comments about the paper, with one maintainer providing an annotated pdf with more detailed comments. Despite their busy schedule and the asynchronous nature of email communication (one cannot force someone to reply), only two maintainers left two or more questions blank. We come back to this in Section 6. The email replies were then analyzed by two of the authors and summarized into a table (Table 5) in order to compare the findings across all 6 maintainers.

At a high level, the obtained feedback showed us whether the activities as a whole made sense, whereas at a lower level it exposed inaccuracies, missed workarounds and any factual errors. We then used this feedback to flesh out the description of the seven documented activities and the 13 challenges, to obtain the final version of the activities documented in the present paper. The contacted members suggested five additional activities, however since we did not have sufficient empirical support for these activities in our data sample, we did not add them to the documented activities. Instead, we discuss those additional activities in Section 6.

Table 3 Overview of integration activities and their prevalence in the three distributions. Activities below the horizontal line were not common enough to be documented.

Activity	Explanation	%Deb.	%Ub.	%Fre.
A. `New Package`	Integrating a new software project.	1.04	0.78	13.54
B. `Upstream Sync`	Updating to a new upstream version.	40.89	43.75	57.81
C. `Dependency Management`	Managing changes to dependencies.	38.80	30.73	28.39
D. `Packaging Change`	Changing a package's packaging logic.	43.49	44.01	38.80
E. `Product-wide Concern`	Enforcing policies across all packages.	4.95	3.13	25.00
F. `Local Patch`	Patching upstream source code locally.	22.40	28.39	12.24
G. `Maintainer Transfer`	Managing unresponsive maintainers.	5.73	0.00	2.86
H. `Security`	Patching a security vulnerability.	4.43	1.30	0.78
I. `Internationalization`	Internationalization of packages.	4.17	1.56	0.26
J. `Other`	Catch-all for rare activities.	2.34	4.95	1.04

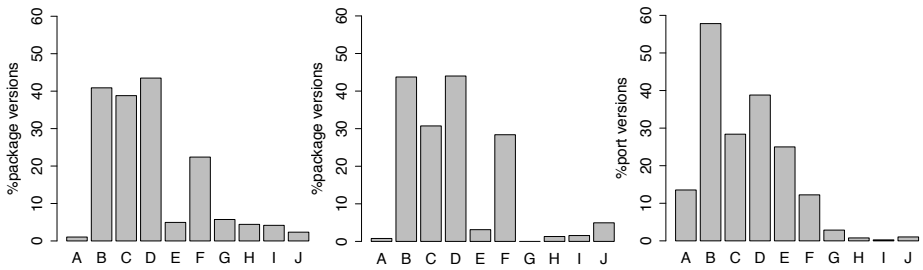

Fig. 5 Popularity of the integration activities of Table 3 in the 384 sampled (a) Debian, (b) Ubuntu and (c) FreeBSD package-versions (confidence interval with length 5% for a 95% confidence level).

4 Integration Activities in Distributions

Table 3 gives an overview and short explanation of the seven major integration activities that we documented, as well as three less common ones. The table also provides the percentage of sampled Debian, Ubuntu and FreeBSD package-versions that involve each of the activities (within a confidence interval of 5%). Those numbers are also plotted on Figure 5. Since a new version of a component can involve multiple integration activities, the percentages in the plots add up to more than 100%. `Upstream Sync`, `Dependency Management` and `Packaging Change` are the most frequently occurring activities in Debian and FreeBSD. `Local Patch` is also common in all three projects, whereas `New Package` and `Product-wide Concern` are common for FreeBSD.

The next subsections discuss each of the seven major integration activities in detail. For each activity, we provide:

Intent Short outline of the goal of the activity.
Motivation Short description of the role and rationale of an activity.
Major tasks The major steps involved with the activity.
Participants A list of stakeholders from Figure 2 involved with the major tasks of the activity.
Interactions Activities that co-occurred substantially with a given activity in package-versions, and hence are related.

Literature Discussion of prior work and approaches for the activity, as well as prevalence of the activity outside the context of OSS distributions.

Notable instances Concrete examples of the activity from the sampled Debian, Ubuntu and FreeBSD package-versions.

A. New Package

Intent: Integrating a previously unpackaged upstream component into a distribution.

Motivation: The users of the distribution or the maintainer of a package require new functionality provided by a component that has been identified but is not yet part of the distribution.

Major Tasks:

1. Recruiting a Maintainer responsible for integrating the new component and for liaising with the upstream project is one of the most important decisions to take [46, 59]. Most commonly, an upstream developer or motivated end-user requests an upstream component to be integrated in the distribution. One of the distribution's maintainers might pick up this request and become the maintainer. Alternatively, the upstream developer can package the component herself and ask a distribution maintainer to "sponsor" this package, i.e., to review and to upload it to the distribution's package repository. In that case, although the majority of the integration is done upstream, the maintainer still has the end responsibility. Another possibility is that the distribution appoints a maintainer to the integration of a new component because of a clear need in the distribution.

2. Packaging an Upstream Project requires access to the project's source code (except for binary-only packages like Adobe Flash) and verification of its license. The maintainer then proceeds to determine the build-time and run-time dependencies of the package. If a dependent component is not yet in the distribution, it has to be packaged first. This is a process of trial-and-error, trying to build the package and fixing any dependency problems. The maintainer might have to customize the software or its makefiles so it would build correctly in the environment of the distribution. When porting the package to other platforms than Linux- or GNU-based ones, it is often needed to remove dependencies on Linux- or GNU-specific libraries or functionality. This can take significant effort. Finally, the maintainer needs to make sure that the package follows the distribution's policies, such as specific locations for configuration files and manual pages.

3. Creating the Package's Metadata. The maintainer is responsible for creating the package metadata like the package name, version number and the list of dependent packages. Such metadata is necessary to add the package to the distribution's package management system ("apt" in Debian/Ubuntu, or the port system in FreeBSD) to enable the automatic and systematic building, packaging, and deployment of the software project.

4. Integration Testing. The package must build and run consistently on all supported architectures. Typically, two rounds of tests are used to verify a package. The first round involves only maintainers ironing out any obvious functionality or platform issues. The second round involves uploading the package to a staging area (e.g., "unstable" in Debian), from where expert end-users can install it for use in their daily work. Bugs identified by these users are reported (together with possible patches) to the maintainer, who incorporates this feedback in a new version of

the package that is re-uploaded. Some distributions, like Ubuntu, have tools to automatically run integration testing and identify integration issues.
5. Publishing the Package. If a staged package contains severe bugs, it might be (temporarily) removed from the staging archive until the bugs are resolved. If the package has been stable for a certain period of time, it becomes eligible for inclusion in an upcoming release. The package is either moved to that release's archive (Debian/Ubuntu), or to the source code repository (FreeBSD).
Participants: maintainer, upstream developer, package community, expert end-user.
Interactions: `New Package` is a prerequisite of the other six activities, and usually occurs by itself (i.e., a package-version only involves `New Package`, and no other activity). In 2.3±5% of the FreeBSD package-versions, it also involves `Local Patch` to fix a bug or to make the package compile.
Literature:
In the context of COTS reuse, additional tasks are involved, especially contract negotiations [7, 65]. Lewis et al. [49] note that "Vendors are driven by profits [...] They can be cooperative and responsive when it is in their perceived interest to be so." Various guidelines and risk assessment tools exist to help companies or federal departments select the right COTS components [7, 49]. They, for example, recommend to find COTS components that fit with the existing architecture, or possibly adjust the architecture first, rather than requiring the COTS vendor to customize their component to the system at hand (since that could be very costly). This is different from OSS distributions, where monetary incentives typically do not exist and OSS distributions sometimes carry enough weight to convince upstream components to adapt to them rather than the other way around.

Although not applicable in the case of packaging organizations like OSS distributions, the identification of COTS/OSS components for reuse is a known challenge as well [64, 79], typically requiring extensive web or literature research, or insightful recommendations by experts. While maintainer recruitment and integration testing are known research problems, the other tasks are less known in research.
Notable Instance:
A `New Package` with customization: *irssi-plugin-otr (Ubuntu)* is an IRC client plugin integrated in July 2008. A first customization changed the location for documentation to the Ubuntu default location. The second customization fixed the package's build process to not download required header files during the build, since the Ubuntu build servers do not have network access.

B. Upstream Sync

Intent: Bringing a package up-to-date with a newer version of the upstream component.
Motivation: As shown in Figure 5, synchronizing the existing packages of a distribution with a newer upstream version forms the core activity of integration. End-users expect package maintainers to update their packages to the latest features and bug fixes as soon as possible, while maintainers are more concerned about the long-term stability of a package.
Major Tasks:

1. Becoming Aware of a New Upstream Release largely depends on distribution-specific dashboards that automatically track the development progress of upstream projects. For example, Debian's watch file mechanism specifies (1) the URL of the upstream project's download page with all releases of a component, as well as (2) a regular expression to identify the source code and a version number for each release. If the highest version number surpasses the current version, this means that a new release is available.

Derived distributions (e.g., Ubuntu) not only need to synchronize with the upstream projects, but also with their own parent distribution, typically at the start of a new release cycle. For example, out of 167 analyzed Ubuntu package-versions involving `Upstream Sync`, 99 versions were synchronized with the upstream project, 65 were synchronized with the parent distribution (Debian) and 3 were synchronized with both. Since the derived distribution can leverage the `Upstream Sync` and other activities performed by the maintainers of the parent distribution, risk assessment (task 2) becomes slightly easier. However, keeping track of which patch was synchronized from which upstream project requires rigorous book-keeping. Projects use custom dashboards for this, sometimes interfacing with the bug reporting infrastructure.

2. Assessing the Risk of an Upstream Release requires the maintainer to review the changes to the previous upstream version [70] in order to estimate whether the new version is production-ready. These changes run the risk of breaking important functionality, while end-users do not always need the new features and bug fixes. Despite the importance of this analysis, in practice it currently is a largely manual task supported by basic tools like "diff" [70], change and commit log messages, email communication with upstream developers, and experience.

The outcome of risk assessment is often to not update to a full new release, but to "cherry-pick" a select number of acceptable changes out of all changes made upstream or by another distribution, then merge those changes into the current package-version (discarding the other changes). For example, an upcoming release of a distribution might be too nearby, making the full import of a new version of a component too risky. Instead, maintainers would cherry-pick the show-stopper bug fixes that they are most interested in. Some distributions, like FreeBSD, prefer not to cherry-pick, i.e., they either take a new version of a component as a whole, or do not update to it.

3. Updating Customization involves revisiting the customizations (patches) performed on earlier versions of the packaged component (e.g., the initial `New Package` or later `Local Patch` activities). Maintainers typically submit these patches upstream, to be merged. As a consequence, some patches no longer need to be maintained locally and can be discarded by the maintainer. Other patches, however, need to be updated by the maintainer to be cleanly applied to the new version of the upstream package. Just like task 2, this requires manual analysis of the patch and the new package-version.

4. Updating the Package's Metadata, cf. task 3 of `New Package`.

5. Integration Testing, cf. task 4 of `New Package`.

6. Publishing the Package, cf. task 5 of `New Package`.

Participants: maintainer and upstream developer.

Interactions: `Upstream Sync` is a pivotal activity that can be accompanied by any other activity, except for `New Package` (by definition). `Upstream Sync` occurs

mostly together with `Packaging Change`, `Dependency Management`, `Local Patch` and (in source-based distributions) `Product-wide Concern`.

Literature:

Together with `Local Patch`, `Upstream Sync` is the most discussed integration activity in literature, independent of the type of reuse (COTS/OSS/ISS) or organization (OSS/commercial) [49, 65], and it is the source of most of the issues related to `Dependency Management` (sometimes even preventing `Upstream Sync` of other packages). For example, Begel et al. [5] report that at Microsoft up to 9% of 775 surveyed engineers rely on other teams to inform them of changes to a component they rely on. Researchers [20, 59] and practitioners [46] recommend to continuously monitor (or inquire) for new versions and their impact on the software system, even appointing a specific gatekeeper responsible for doing this. This also helps mitigate one of the largest risks of reuse: the component vendor going out of business [49].

Since reuse induces a dependency on the provider of a COTS/OSS/ISS component (who fully controls the component's evolution [49]), researchers have reported two extreme approaches to deal with this dependency: swiftly updating to each new component version [11, 54, 79] versus sticking to a particular version and patching it for the organization's particular needs [54, 59, 71]. There is no systematic methodology to decide between the two approaches and hybrid approaches in between like cherry-picking [49], typically personal experience is the deciding factor [59], while other factors like the safety-critical nature of a software system can play a role as well [49]. Interestingly, many integration issues could in fact be avoided if the new component version would be backwards compatible with the previous version [16, 79], but this is outside the control of the organization that reuses a component.

Notable Instances:

A low-risk `Upstream Sync`: *Gnash (Ubuntu)* is a Flash player that was updated to upstream version 0.8.7 in March 2010 (#522254[10]), right at the start of the Ubuntu feature freeze window (i.e., close to the next release). Since new features are technically not allowed in a freeze window, a member of the Ubuntu release team needed to explicitly approve the `Upstream Sync`. As Gnash is a package inherited from Debian, and the update mostly contained bug fixes, version 0.8.7 quickly got synced.

An `Upstream Sync` taking a long time: *Krita 2.1.1-1 (Debian)*, the painting program of the KOffice suite, was broken early May 2010 because one of the libraries it depends on (libkdcraw7) had been replaced by a newer version (libkdcraw8) in an `Upstream Sync` of KDE 4.4.3 (#580782). Unfortunately, the solution (an `Upstream Sync` to KOffice 2.2.0), took 2 months because this new version of KOffice introduced too many new functionalities, requiring the package to be tested more thoroughly.

A patch cherry-picked from another distribution: *libpt 1.10.10 (Ubuntu)*, a cross-platform library, relied on the new gspca webcam driver provided by the 2.6.27 Linux kernel. For this driver to work, all programs and libraries consuming the webcam stream now had to load the libv4l wrapper libraries at run-time, forcing 62 Ubuntu packages to be modified. Since three weeks earlier a patch had been

[10] This notation refers to a bug report in the distribution's bug repository.

uploaded to Fedora (another distribution) to make these changes for libpt, this patch was cherry-picked into Debian (and Ubuntu).

C. Dependency Management

Intent: Keeping track of the dependencies of a package to make sure it can be properly built and run.
Motivation: Packages depend on other packages to be built (e.g., compilers and static libraries) and to be run (e.g., dynamic libraries and services). For example, in our data set, Debian packages containing dynamic libraries have on average 6.4 packages depending on them directly (median: 2.0), and 47.6 transitively (median: 3.0). If a package on which many other packages ("reverse-dependencies") depend changes, for example because of an `Upstream Sync`, that change might break its reverse-dependencies.

A special case of such a change are "library transitions", i.e., changes to the public interface of a shared library that might force dozens of packages to be rebuilt or, in the worst case, to be adapted to the new interface via source code changes. For example, if the C runtime library would change, all packages using C might need to be changed and/or re-built.
Major Tasks:
1. Becoming Aware of Dependency Changes either happens automatically (see `Upstream Sync`), or based on an announcement by the maintainer of a dependent package that is about to change significantly. The latter announcement typically is sent to the release manager and any affected maintainers, leaving time to discuss the repercussions of the update. In case such an announcement has not been done, at the very minimum, the maintainer should notice a change in the API through the updated interface version ("SONAME") of a dynamic library[11]. For example, a dynamic library "libfoo" with interface version 1 would have a SONAME of "libfoo.so.1". If this SONAME suddenly changed to "libfoo.so.2" upstream, maintainers would know that the API of the component has changed substantially.
2. Assessing the Risk of a Dependency Change is similar to task 2 of an `Upstream Sync`: determining which and whose packages broke because of a change is largely a manual task, requiring insight into how an API is used by other packages, whose implementation and algorithms are typically unknown to the maintainer. Unfortunately, no tool support is available in practice to assist in this task. Typically, the build logs are checked for errors and the package is driven through a small smoke test scenario.
3. Fixing the Damage either happens atomically, i.e., the changed package and all its reverse-dependencies are updated at once (FreeBSD), or interleaved, i.e., each of the packages is updated independently (Debian/Ubuntu). Atomic updates can delay a new package-version as long as not all broken packages have been updated successfully, but at least the end user will not be impacted by inconsistent packages. Distributions like Fedora and Ubuntu use sandbox build environments to atomically update a transitioning library with all its reverse-dependencies in isolation, without affecting other packages (and hence users) [81].

[11] If the maintainer finds out that the interface did change without a SONAME update, she would contact upstream to ask for an update of the SONAME, then perform an `Upstream Sync` of the updated library before resuming the `Dependency Management` of the library's reverse-dependencies.

Whether or not the update model is atomic, the maintainer of the library causing the changes is responsible for performing all rebuilds. The maintainer analyses the build and test logs to determine which packages failed to build, and attempts to write patches for those, using her knowledge of the API changes. If this fails, she needs to assist the failing packages' maintainers to resolve the transition issues, similar to delivery advocates for ISS reuse [79]. To keep track of which packages have already been re-built, the release manager and maintainers use a tracking system: Ubuntu and Debian both use a custom library transition tracker, while Ubuntu sometimes uses a bug tracker.
4. Updating the Packages' Metadata, cf. task 3 of `New Package`.
5. Integration Testing, cf. task 4 of `New Package`, once the whole transition is complete (atomic model) or for each updated package separately (interleaved model).
6. Publishing the Package, cf. task 5 of `New Package`.
Participants: maintainers of the changed package and those of its reverse-dependencies, release manager.
Interactions: `Dependency Management` can be accompanied by any other activity, except for `New Package`. It occurs mostly together with `Upstream Sync`, `Packaging Change`, `Local Patch` and (in source-based systems) `Product-wide Concern`.
Literature:
Similar to `Upstream Sync`, `Dependency Management` is independent of the kind of reuse and organization. Begel et al. [5] observed a wide range of mitigation techniques for dependency problems at Microsoft, ranging from minimizing the number of dependencies to explicitly planning backup strategies to deal with dependency issues. Other companies, such as the one studied by de Souza et al. [19,20], stressed the importance of vendor-integrator communication to reduce the effort required for "impact management" of reused APIs. Managers first should build an impact network consisting of people affecting or affected by their component, then use frequent email communication or people assigned explicitly to a particular API (or ISS component [79]) to manage forward (i.e., on other teams) and backward (i.e., on their team) dependency impact. Similar to other major companies like Google [91], as well as the studied OSS distributions, a team is required to inform its clients of major API breakage. de Souza et al. [20] note, however, that one should not forget the ripple effect of "indirect" (i.e., transitive) dependencies.

Similar to `Upstream Sync`, backwards compatibility of dependent packages can avoid many integration issues [16, 79]. Furthermore, many `Dependency Management` issues are due to unnecessarily high coupling between components by relying on implementation details [78] and private APIs [79]. Hence, using components via explicit [79] and stable [59] interfaces can avoid many problems. Finally, packaging organizations like distributions can eliminate many dependency issues of their users by providing assemblies (sets) of integrated components instead of individual components. This is why many distributions offer so-called "virtual" packages, for example to integrate all core packages of Perl, KDE or GNOME.
Notable Instances:
A surprise library transition: A library interface change to the *libfm 0.1.14-1 (Debian)* file manager library was not announced by the upstream developer. As a consequence, applications built against the old version of libfm ("libfm.so.0"), such as the pcmanfm file manager, broke (#600387). The dynamic linker had no way of knowing that "libfm.so.0" was no longer the original library version all

packages were built against, but rather the new version with a different interface that should have been named "libfm.so.1".

Problems with non-atomic fixes of dependency changes: The transition of *Perl 5.10 (Debian)*, the Perl programming language ecosystem, to Perl 5.12 at the end of April 2011 (#619117) took slightly over two weeks, during which over 400 packages (directly or indirectly depending on Perl), including high-profile ones such as vim, subversion, rxvt-unicode and GNOME, were not installable from the staging area until all their dependencies were rebuilt consistently against Perl 5.12.

A dependency change requiring only a rebuild: The chances of acceptance for *Boost 1.34.1 (Ubuntu)*, a general-purpose C++ library, in Ubuntu 7.10 looked slim, since Ubuntu had just entered its "Feature Freeze" (only bug fixes were still accepted for the upcoming release) and all Boost's reverse-dependencies had to be updated. However, the contributor championing the new Boost release was able to convey the urgency of the release (fixes to show-stopper bugs) and the package maintainer verified that all reverse-dependencies could just be rebuilt without source code changes.

D. Packaging Change

Intent: Changing the packaging logic or metadata to fix packaging bugs, follow new packaging guidelines or change the default configuration, either for binary or source packages.

Motivation: The packaging process combines the build process [58] of an upstream component with the dependency management and packaging machinery of a distribution. Hence, understanding the packaging process is not a trivial process, and bugs slip in frequently. Furthermore, as the packaged component evolves, its packaging requirements evolve as well. For example, new features might have been added that need to be configured in the package. The `Packaging Change` activity covers any such changes to the packaging, building and installation logic and metadata of a package.

Major Tasks:

1. Replicating Reported Problems is a prerequisite in order to fix a packaging problem. Ideally, the maintainer would like to clone the packaging environment of a bug reporter, or at least have a complete description of the build platform, all installed libraries and their versions. Tools exist to generate such a description when submitting bug reports, yet inexperienced bug reporters often do not know or forget to use those.

2. Understanding the Build and Packaging Process is a necessity in order to be able to fix packaging bugs or enhance the packaging logic. Such understanding currently is based on interpreting the build and execution logs of packages. Furthermore, trial-and-error is commonly used when changing the packaging logic. Since there is no dedicated way to test build and packaging changes, the maintainer verifies the correctness of those changes by manually installing the package and running the unit or user tests of the package.

3. Integration Testing, cf. task 4 of `New Package`.

4. Publishing the Package, cf. task 5 of `New Package`.

Participants: maintainer, package community (for testing), expert end-user.

Interactions: This activity is performed during most of the other activities, such as `New Package` and `Upstream Sync`. Frequently, this activity requires a `Local Patch`.

Literature:
The `Packaging Change` activity has not been discussed thoroughly in prior research, except for the well-known difficulty of configuring COTS/OSS/ISS components [79]. Such configuration issues are due to the fact that, by default, components need to be generic and contain many features, whereas a specific integrator only needs some of those. The need to adapt packaging logic is specific to the domain of packaging organizations (of which OSS distributions are a subset), since they are a mediator between upstream components and final users, and hence require upstream components to fit into their own package management system.
Notable Instances:
A package with missing files: The librt shared library implementing the POSIX Advanced Realtime specification had been dropped without warning from the GNU standard C library on Debian (*libc6 2.3.6-18*), breaking the XFS file system package (#381881). To resolve this case of `Dependency Management` for XFS, a `Packaging Change` was made to libc6's package metadata to indicate that librt was no longer provided.
Broken packaging because of changed guidelines: Versions 2.6 to 3.2 of *Python (Ubuntu)*, the Python programming language ecosystem, suddenly failed to build on Ubuntu (#738213) because essential libraries like libdb and zlib on which python depended could not be found anymore on the build platform. The change in directory layout was a result of the work on enabling 32 and 64 bit versions of libraries to be installed on a single machine.
Broken packaging because of upstream changes: The *GNU Octave (FreeBSD)* developers changed the layout of their web site as well as the build logic of some of their projects (#144512). The maintainer had to fix the code fetching script and refactor the existing build script shared by all GNU Octave ports into separate scripts for the individual ports.

E. Product-wide Concern

Intent: Applying product-wide policies and strategic decisions to the integrated packages.
Motivation: Since a distribution integrates thousands of packages, there are important rules and strategic decisions that should be followed in order to make the distribution coherent and consistent. For example, a new standard for package help files should be adopted by all packages, either all at once or at their own pace. Similarly, strategic decisions to transition to a new version of a core library or to move to a new default window manager should be followed up as uniformly as possible by all involved packages.
Major Tasks:
1. Determining Ownership and Timing of Changes happens through discussions between the co-ordinator (release manager or a volunteer) of the product-wide concern and the affected maintainers. The co-ordinator notifies all affected package maintainers about the decision, explaining the motivation of the `Product-wide Concern`, the end goal and the different steps involved in getting there. Those steps depend on the enforcement strategy in use.

2. Enforcing the Concern happens either through centralized or distributed enforcement. With centralized enforcement, the `Product-wide Concern` co-ordinator applies the concern's changes herself on all affected packages at once. Maintainers only need to test if their package still works and report a bug if it does not. With distributed enforcement, the package maintainers, briefed by the co-ordinator, are in charge of the change for their own package. This gives them the freedom to implement a `Product-wide Concern` as they see fit, but might delay updates to their packages' reverse-dependencies. While the concern is being enforced, the co-ordinator continuously monitors the status of the concern via dashboards, mailing lists and/or bug reporting systems.

Debian uses distributed enforcement, FreeBSD uses centralized enforcement and Ubuntu uses both. Derived distributions like Ubuntu automatically leverage `Product-wide Concern` changes performed by the contributors of the parent distribution. FreeBSD co-ordinators use regular expressions to change the packaging logic of hundreds of ports at once, thanks to the strict naming conventions in the packaging logic. Given the high risk of such product-wide changes in FreeBSD, the co-ordinator needs approval by the release manager, after which the whole distribution is rebuilt on the distribution's build cluster to check the effects of the product-wide change.

3. Integration Testing, cf. task 4 of `New Package`.

4. Publishing the Package, cf. task 5 of `New Package`.

Participants: maintainer, co-ordinator, release manager.

Interactions: `Product-wide Concern` is typically accompanied by `Dependency Management`, `Upstream Sync` or `Packaging Change`.

Literature:

Similar to `Packaging Change`, `Product-wide Concern` is a relatively unknown activity. For example, Curtis et al. [17] identify the issue that "Projects must be aligned with company goals and [that they] are affected by corporate politics, culture, and procedures", and they stress that the "inter-team groups dynamics" (between an integrator and upstream) significantly complicates the already complex "intra-team group dynamics". However, no concrete advice or discussion of the tasks involved are provided, especially not in the context of multi-component integration at the scale of OSS distributions (thousands of integrated components).

Notable Instances:

***The massive migration to* GCC 4 (Debian)** in July 2005 is an example of a `Product-wide Concern` with distributed enforcement. Since the compiler suite broke C++ programs compiled with earlier GCC versions, all C++ packages using GCC had to be rebuilt. An approach typically followed in cases like this[12,13], is to (permanently) rename the packages after rebuilding by attaching a suffix like "+b2". This ensures the visibility of rebuilt packages, enabling other packages to explicitly depend on the rebuilt versions.

The migration to* Dash *as the default command shell in Ubuntu 6.10 (October 2006) and Debian Lenny (February 2009) illustrates the differences between centralized and distributed enforcement. The Ubuntu co-ordinator instantaneously made Dash the default shell, breaking many packages' scripts and build files (centralized). Although several users were enraged, the co-ordinator consistently re-

[12] `http://bit.ly/FOCJHf`

[13] `http://lwn.net/Articles/160330/`

ferred to the maintainers and upstream developers of the failing packages to fix incompatible Bash-specific code ("bashisms"). A web site with official migration strategies and workarounds was provided.

When Debian discussed their move to Dash (independently from the Ubuntu move)[14], the Ubuntu co-ordinator convinced them about the importance of clear release goals and communication with all stakeholders. The Debian developers then built tools to screen all packages for known bashisms. Maintainers of packages containing bashisms were notified by email and requested to fix the bashisms by a certain date (distributed).

F. Local Patch

Intent: Maintaining local fixes and/or customizations to a package.
Motivation: Integrators and their users will find bugs in packages. Some of these bugs are package-specific, while others are due to the integration of the package in the distribution. Typically, maintainers are encouraged to send the fixes for both kinds of bugs upstream, such that the upstream project will take ownership of the code (and its maintenance) and include it by default in their project. In practice, however, many integration bug fixes are not accepted by upstream (or take time to be adopted) and tend to end up as local patches that need to be maintained by the integrator and re-applied by the integrator upon each `Upstream Sync`. The same holds for customization changes specific to a distribution, for example because of `Product-wide Concern`.
Major Tasks:
1. Getting a Local Patch Accepted Upstream requires a patch that fixes the bug in a clean way and follows the programming guidelines of the upstream developers. After thorough testing, the maintainer submits the patch to the preferred bug reporting system of the upstream project. The report should be as detailed as possible, making clear what bug is fixed, in which version of the project, and what the impact is on the users of the distribution. Either the patch is accepted in a reasonable period of time, or it is not. If accepted, the maintainer can discard his `Local Patch`. Otherwise, the maintainer is responsible for maintaining and re-applying the `Local Patch` across all future versions of the package.
2. Maintaining the Patch upon an `Upstream Sync` is the maintainer's responsibility until the `Local Patch` is accepted by upstream (if ever), cf. task 3 of `Upstream Sync`. As such, `Local Patch` is a very common activity, involving $22.1 \pm 5\%$ (Debian), $28.4 \pm 5\%$ (Ubuntu) and $12.2 \pm 5\%$ (FreeBSD) of all package-versions. Of these versions, only $7 \pm 5\%$ (Debian), $0.3 \pm 5\%$ (Ubuntu) and $0 \pm 5\%$ (FreeBSD) had to update an existing `Local Patch`, whereas $24.7 \pm 5\%$ (Debian), $11.9 \pm 5\%$ (Ubuntu) and $6.3 \pm 5\%$ (FreeBSD) could stop maintaining the `Local Patch` because it was included into a new upstream version. To keep track of local patches, Debian-based distributions use patch management systems such as "quilt", "dpatch" and "git", while FreeBSD maintainers manage patches manually.
3. Updating the Package's Metadata, cf. task 3 of `New Package`.
4. Integration Testing, cf. task 4 of `New Package`.
5. Publishing the Package, cf. task 5 of `New Package`.
Participants: maintainer, upstream developer, bug reporter.

[14] `http://bit.ly/z3ORxT`

Interactions: `Local Patch` is typically accompanied by `Upstream Sync`, `Packaging Change`, or `Dependency Management`.

Literature:

The paradox of on the one hand having to submit a patch upstream to avoid maintenance but on the other hand having a hard time getting the patch accepted, is the most studied integration challenge in the literature, across different kinds of reuse and organizations [3,11,59,78,79]. No silver bullet exists, although, similar to `Upstream Sync` and `Dependency Management`, close collaboration of an organization with the upstream project is generally recommended [79], even in the case of COTS [64]. However, such a collaboration takes a lot of time, effort and goodwill, and also does not guarantee that the upstream project will accept and maintain the patch [90]. In fact, it often happens that even an accepted patch still needs to be maintained by the downstream organization (since the organization has the required expertise) [45].

An opposite approach has been successful in the case of ISS, where the ISS team reaches out to the teams that reuse its components to help them with integration [79]. Alternatively, one could use COTS-style glue or wrapper code to avoid changing the actual code altogether [26, 54]. However, such approaches are less powerful (one loses the benefits of OSS/ISS) and still require maintenance. As a kind of middle ground, many organizations use packaging organizations like OSS distributions as a maintenance buffer between upstream and themselves [59], shifting the problem to the distributions. In the presence of sufficient industrial partners, one could even consider making an independent fork of an upstream component, but this is quite costly and in the end not that successful in practice [90]. Note that patches for local usage or configuration will never be picked up upstream, hence require eternal maintenance. This applies especially to end-users, who might have local patches on top of a distribution's package.

Notable Instances:

A patch that is quickly adopted upstream: The Debian and Ubuntu packages of the *GNOME sensors-applet (Debian/Ubuntu)* desktop widget for temperature and other sensors featured "ugly, outdated icons" (#69800) because the newer icons did not comply with the license policy of Debian and Ubuntu. To fix this, the Ubuntu maintainer built a local patch on top of the Debian package to use the newer icons in Ubuntu, while the upstream developer contacted the icon designer to make the new icons compatible with Debian by adding an additional license to the icons (an example of the "Disjunctive" legal pattern [35]). The designer complied, and the Ubuntu maintainer reported the license change to the Debian maintainer, such that he could drop his `Local Patch`.

A* `Local Patch` *can cause havoc: A notorious security hole in the *OpenSSL* Debian package (an implementation of the SSL/TLS protocols) was introduced into Debian by a local patch and lasted from May 2006 until May 2008[15]. A call to the function adding randomness to a cryptographic key had accidentally been commented out by a `Local Patch` (#363516). The Debian maintainer had contacted upstream, but did not fully disclose himself, nor his plans, and was largely ignored[16]. The patch was never sent upstream for inclusion afterwards. To complicate the issue further, the address of the mailing list contacted by Debian was

[15] `http://lwn.net/Articles/282038/`

[16] `http://bit.ly/w7rn04`

not the real OpenSSL development list, since that one was hidden from non-developers[17]. This security hole propagated to over 44 derived distributions, without any of the maintainers or contributors involved identifying the bug.

G. Maintainer Transfer

Intent:

Maintaining a package if the maintainer is absent, unwilling or incapable to further maintain a package.

Motivation:

Being a package maintainer is a major responsibility, since it requires mediating between upstream projects and the end-user, typically for multiple packages at a time. However, maintainers may have periods during which they cannot spend the required time on integration, they may lose interest in certain packages, or they could just become unresponsive to bug reports or user requests. In the worst case, a package could even be orphaned when the maintainer quits. To prevent packages (and any product based on it [54]) from stalling, OSS distributions need to provide a means to keep packages evolving, while bypassing or overriding a maintainer.

Major Tasks:

1. Overriding the Maintainer depends on how a distribution organizes package ownership. If package maintenance is shared across all distribution developers collectively, the concept of overriding a maintainer is not relevant. In Ubuntu, for example, packages in the commercially supported Main and Restricted archives are managed by a team known as Core Developers, whereas the packages in the commercially unsupported Universe and Multiverse archives are supported by the community under the guidance of a team known as "Masters Of The Universe" (MOTU). Any developer can modify any package, as long as it is managed by the developer's collective and the change does not introduce unnecessary divergences compared to upstream. In case of disagreement amongst developers, there are conflict resolution procedures in place, but those rarely need to be used.

Distributions with individual package ownership, on the other hand, need a `Maintainer Transfer` policy to take over the role of a maintainer if she becomes unresponsive or disappears altogether. A contributor proposing an `Upstream Sync`, `Dependency Management`, `Infrastructure Change` or a `Local Patch` that fulfils certain criteria can explicitly mark her change as a `Maintainer Transfer`. In Debian, for example, this is called a "Non-Maintainer Upload" (NMU), and is only valid for changes that fix an important, known bug. Debian provides the "nmudiff" tool to help contributors submit NMUs.

The unique property of a `Maintainer Transfer` change is that a timer is attached to it, with a delay depending on the severity of the proposed change (e.g., FreeBSD typically uses a delay of 2 weeks). Unless the maintainer replies to the change on time, the change is set to go in automatically once the timer expires. If the maintainer replies on time, she can request suspending the timer in order to review the change. If not approved, the contributor needs to revise the change corresponding to the maintainer's comments.

We found that 5.7±5% (Debian) and 2.9±5% (FreeBSD) of all package-versions contain an instance of `Maintainer Transfer` (Ubuntu has collective package own-

[17] `http://www.links.org/?p=327`

ership, hence does not have such transfers). The min/median/max number of days until such changes were accepted is 0/1.5/556 days for Debian and 1/16/465 days for FreeBSD. In Debian, the median value is very low, indicating that maintainers often commit a `Maintainer Transfer` before the timer goes off. In FreeBSD, time-outs are much more common. The cases with maximum time-out in Debian (#325110) and FreeBSD (#140303) correspond to packages that temporarily were orphaned, i.e., the maintainer officially stepped down.

2. Supporting Orphaned Packages is typically done by an ad hoc team of volunteers, based on casual contributions or reported critical bugs. In Debian, the QA team typically jumps in to make changes to orphaned packages.

3. Adopting Orphaned Packages either happens by volunteers interested in an orphaned package, or by convention, when a contributor provides patches for an orphaned package and automatically becomes the new maintainer. For example, if no feedback is received for a patch in FreeBSD within three months, the maintainer is deemed to have abandoned the package and any contributor may assume maintainership [82, Section 5.5].

Participants: maintainer, contributor.

Interactions: `Maintainer Transfer` can co-occur with all other activities, except for `New Package`.

Literature:

We could not find any reference to the `Maintainer Transfer` activity in literature. However, Curtis et al. [17] and Lewis et al. [49] do stress the importance of having "system-level thinkers" as maintainers, who are able to sufficiently understand both the specific domain of the integrated component as well as the overall architecture of their own system. According to our analysis, the `Maintainer Transfer` activity would kick in as soon as the maintainer of a component would not possess those skills.

Notable Instances:

An NMU helping out a busy maintainer: *httrack 3.40.4-3.1 (Debian)*, an offline browser, fixed an issue with the file system locations for test files. The bug was reported on the 11th of October 2006, followed one week later by a proposed NMU by a contributor. A couple of hours later the NMU was approved by the maintainer, who noted (#392419): "Thanks a lot, I didn't yet had [sic] the change [sic] to review the issue".

An NMU with strings attached: The maintainer of *libcdio 0.78.2+dfsg1-2.1 (Debian)*, a library for accessing CD media, had been warned on the 20th of January 2008 about C++ header file issues with the upcoming release of GCC 4.3 (`Product-wide Concern`). Two months later, a contributor sent in an NMU patch fixing the compiler errors. One day later, the maintainer chimes in (#461683): "I don't object to a NMU (I know I haven't been handling my libcdio package in the best possible way), but if you wish to NMU, please consider applying the patches that were sent to other bug reports"). The NMU was approved the same day.

A hostile NMU: On the 18th of May 2007, a contributor requested an `Upstream Sync` to the new upstream release (1.3.2) of *libjcalendar-java 1.2.2-6.1 (Debian)*, a calendar picker component, and also proposed a `Packaging Change` to support the Kaffe Java VM. However, since nothing happened for one week, the contributor added a comment to both bug reports stating "I am planning a NMU if nothing happens (again)" (#424981, #424982) . The next day, the maintainer replied (#424981) "I admit that I'm not very reactive, but before you do your NMU,

Table 4 Open challenges for integration activities.

Area	Challenge
packaging	· insight into upstream build process · automatic build-/run-time dependency extraction · accurate replication of packaging environment
testing	· cross-platform testing of package & its dependencies · integration testing during packaging · accurate replication of functionality issues
evolution	· determining best moment for `Upstream Sync` · insight into upstream changes · recommendations about important API changes · management of ownership of package changes
merging	· prediction of integration defects · identifying opportunities for cherry-picking · insight into merge status of `Local Patches`

have you checked that Jcalendar 1.3.2 is backwards compatible with version 1.2?". Nothing happened for 1.5 months, until the NMU timer had expired and the NMU went in.

5 Identified Integration Challenges

The seven discussed integration activities document the complexity of integration. Even in the simplest case, i.e., black box integration, maintainers still need to package the integrated project (`New Package`), verify if the integrated product is compatible with each `Upstream Sync`, and follow up on `Dependency Management` changes like library transitions. In the case of white box integration, the integrated projects need to be customized or fixed with `Local Patches`, and streamlined to product-wide policies (`Product-wide Changes`). All the time, the packaging logic and configuration files need to be kept up-to-date (`Packaging Change`), and maintainer activity needs to be monitored (`Maintainer Transfer`).

To paraphrase Curtis et al. [17], we "are not claiming to have discovered new insights" for OSS integration, instead we identified and documented the core integration activities that the maintainers of three large OSS distributions perform on a daily basis "to help identify which factors must be attacked to improve" integration. Although distributions have guidelines on how to address some of these activities [22,82], the differences in terminology (e.g., "NMU" vs. "time-out") and technical procedures (e.g., centralized vs. distributed `Product-wide Concern`) make it confusing to understand and compare the activities, or to study possible tools and techniques to improve these activities. Hence, the unifying vocabulary that we provide is key to understand the integrating process of upstream components, complementing existing work on code integration [15,23,29,67,68] and on selection of reusable components [6,13,50]. Finally, we also compared the activities to those in prior work, in particular in commercial settings.

Throughout our analyses and the documentation of the 7 integration activities, we distilled 13 concrete challenges summarized in Table 4, across four different research areas. Most of the challenges have been discussed earlier in this paper. Ubuntu and Debian are currently in the process of designing an automatic unit and

integration testing system for the packaging process. Similar to defect prediction work at the code level, prediction of integration defects and the effort involved with fixing these defects would be extremely useful. There is some initial work on this [63, 94], but more work is needed to bring such techniques to practitioners. Similarly, a kind of bugzilla repository for managing ownership of changes, i.e., who should update reverse-dependencies, who should perform a `Product-wide Concern` or who should act on an NMU, is needed to improve communication across all involved parties. Insight into the upstream build process [1, 85] currently relies on manual tracing and analysis of build and run-time logs, with only some packages having rudimentary scripts for checking runtime dependencies. In general, however, the ability to accurately replicate bugs in code and build is missing. Packaging environments can vary widely between users, with certain combinations of package and distribution versions causing subtle packaging or run-time problems. Current bug reporting tools automatically include detailed platform information, yet such information is often insufficient to identify `Dependency Management` changes.

As the above challenges impact even three of the largest and most popular OSS distributions, more powerful tool and process support is essential for most of the OSS integration activities, complementing the mailing lists, bug repositories, and custom dashboards (for example to track library transitions) currently in use by organizations. Until now, researchers have only been studying some of the challenges, such as API changes [18] and merge defects [12, 75]). Clearly, more research is needed to support maintainers in the field.

6 Evaluation

The six contacted maintainers pointed out some small factual errors in an earlier version of the documented integration activities, and recent advances (e.g., regarding the automatic test systems being built for Debian and Ubuntu). However, no fundamental errors were identified, nor was ant activity discarded. The identified inaccuracies have been fixed in the activity descriptions above.

Regarding the completeness and usefulness of the documented activities, Table 5 summarizes the replies of the six contacted maintainers. As explained in Section 3.6, two maintainers (M2 and M5) provided empty replies for at least two questions, while M1 left one question open. Hence, we obtained some empty replies for Q2, Q3 and Q5. We now discuss each question's answers.

Q1. What activities did we miss? Five of the maintainers pointed out missing activities, although many of them were captured in some form.

A. "Upstream Lobbying" was in fact mentioned as part of `Local Patch`, but M4 found that it deserved its own activity. Interestingly, M6 mentioned the inverse kind of lobbying, i.e., lobbying in derived distributions for newly reported or fixed bugs. Instead of splitting up `Local Patch`, we decided to keep this activity as is, but add more detail about the lobbying part.

B. "Post-release Maintenance" was suggested by M4 and M2 as a dedicated integration activity encompassing all the activities occurring after a new package-version has made it into a new release of the distribution. M4 notes that "while the maintainer isn't required to support the use of a product they [sic] are often the

Table 5 Maintainer feedback on the usefulness and completeness of the documented activities and challenges.

	M1	M2	M3
Q1	license/copyright analysis	vulnerability resolution post-release maintenance	no
Q2	people unfamiliar with topic	<no reply>	major activities ... ... in easy-to-read way
Q3	<no reply>	<no reply>	more detail/examples
Q4	license tracking	none	none
Q5	DEP5/CDBS license checking autom. dep. checking	<no reply>	automated testing autom. dep. checking
	M4	M5	M6
Q1	upstream lobbying post-release maintenance	package end-of-life	monitoring downstream ... ... distributions for bugs/patches
Q2	useful overview do we document our activities?	<no reply>	nice intro to what ... ... being distro dev is about
Q3	what to do?	nothing	none
Q4	timely integration desktop vs. enterprise hundreds of variants	none	monitoring the status ... ... of all packages ... ... in the distribution
Q5	good question :-)	<no reply>	improvements to package process atomic package updates

first person contacted if someone can't get to build on FreeBSD". Our activities do not capture this activity by itself, only its outcome, for example in the form of a `Packaging Change` or `Local Patch`. This is because many emails could be exchanged regarding a maintenance problem without a corresponding change log item or bug report (i.e., our data set does not capture such discussions). Although this hints at less important integration issues (since they did not need to be fixed or acted upon in some form), future work should analyze the mailing list data of the distributions to uncover this part of the integration work.

C. "License/Copyright Analysis" was mentioned by M1 as an important activity: "copyright/licensing analysis isn't mentioned anywhere, yet it's often a tiresome process when creating a new package (and often forgot [sic] to update on upstream sync)". License analysis did not occur very often in our data set, for example in our Ubuntu samples we only found one occurrence (version "0.4-0ubuntu1" of package "branding-ubuntu"), in which case the license of some files had not been specified as being GPL. For this reason, the activity is captured in our `Other` category.

D. "Vulnerability Resolution" was pointed out by M2 as a missing activity, i.e., the steps performed to address a vulnerability in a timely manner after release. Although it is not one of the top 7 activities (and hence not documented in detail by us), vulnerability resolution occurred relatively often (Table 3), occurring in $4.4 \pm 5\%$ (Debian), $1.3 \pm 5\%$ (Ubuntu) and $0.8 \pm 5\%$ (FreeBSD) of all package-versions. Our data shows how most of these vulnerabilities were reported and fixed upstream. Similar to `Upstream Sync`, distributions first have to become aware of vulnerabilities, then update their packages as soon as a fix is available.

For this reason, vulnerability changes tend to use NMUs (see `Maintainer Transfer`), since the security team wants to update a vulnerable package as soon

as possible, overruling the maintainer if necessary. Often, vulnerability fixes are cherry-picked, leaving other upstream changes until the next official `Upstream Sync`. For example, *cups-base revision 1.44 (FreeBSD)* (24th of January 2005) fixed a vulnerability in the Cups printer server identified and reported upstream by a university student, while *php4 4:4.4.0-3ubuntu1 (Ubuntu)* cherry-picked 8 upstream vulnerability fixes for the php programming language (19th of December, 2005). Since the full details of vulnerabilities and how they were processed internally are not available in publicly available databases, and since it is less common than the seven documented activities, detailed analysis of this integration activity is future work.

E. "Package End-of-life" was a missing and often overlooked activity according to M5. Some packages lose user and maintainer interest over time, hence when the distribution evolves and integration activities need to be performed on the package, either nobody steps up or substantial effort is required by other maintainers to keep the package up-to-date. Similarly, if an older version of a library is rendered obsolete by a newer one, or the older version starts to create conflicts with the newer one, the older version needs to be removed from the distribution. However, we did not find evidence of this activity in our data samples. Our `Maintainer Transfer` activity comes closest, since this one occurs when an unmaintained package is "saved" from end-of-life by a new maintainer.

Surprisingly, the `Internationalization` activity, which is the ninth most frequent activity that we found (Table 3), was not mentioned by any maintainer. This activity comprises all the work related to translation and adaptation of a package to other cultures (e.g., different currencies) [93]. Since distributions reach significantly more users than an individual upstream project could reach on its own, a packaged project has a higher chance of being used in non-English locales. Hence, distributions typically have dedicated teams addressing the internationalization needs of their packages.

For example, the debian-l10n-english team works on the translation templates of packages to facilitate the job of translators (who are often not software engineering experts). Distributions typically solicit `Internationalization` patches once development has been frozen, i.e., the basic new functionality has been stabilized and only bug fixes are still allowed. Although `Internationalization` changes are typically harmless, they can in rare cases keep packages from executing. In January 2006, for example, an incomplete Japanese character prevented the xchat IRC client of FreeBSD from executin. A 1-character fix in a translation template fixed this issue.

Q2. What can the documented activities be used for? M1, M3 and M4 agree that the documented patterns provide a clear overview of the major integration activities, which is useful for novices (M1) as well as any stakeholder involved in integration (M3/M4). M4 noted that the activities do not necessarily need to be used as direct documentation. They could also be used to check how well the distribution collects data or monitors the progress of each integration activity. M3 informed us that the structured, accessible explanations of the major integration activities piqued the interest of two of his package testers, which he believes to be a success. M6 recommended us to "reach out to developers communities with this documentation. E.g., you could write a blog post providing an introduction

to your paper, targetted at distribution devs". We are planning to follow up on this suggestion.

Q3. What is missing from the documented activities? M3 was interested in getting more details and examples for each activity, while M4 wanted to know what the recommended practices and tools for each activity are. Our documented activities on purpose describe only the major tasks and how they are implemented in the three considered distributions, without a dedicated section for "best practices". Given the many challenges identified in Section 5 as well as in Section 2, many activities rely on manual work, and hence do not yet have best practices.

Q4. What challenges did we miss? M1 again mentioned license tracking. M4 noted that the largest challenge is not how to perform each activity, but how to perform them *on time*. Given the ever shorter time frame in between releases [44,69,76], this is indeed an important constraint on the identified challenges. Furthermore, the right activity to do on a particular moment also depends on the end-user: "desktop users want updates ASAP while enterprise users don't want to change their software for multiple years". This echoes known phenomena such as Microsoft's monthly "patch Tuesday" [48] and Mozilla's extended support releases for companies [47]. M4 concluded by warning for the challenges represented by the hundreds of variations in build systems, versioning schemes, projects, etc. Slightly related to this, M6 noted that "something orthogonal is the management of a large amount of software packages: getting a global overview from their status is not easy". This ties into the management-related challenges of Table 4 identified from our data.

Q5. What promising tools/techniques do you see coming up to address some of the challenges? Both M1 and M3 expect automated dependency checking tools to become mainstream, i.e., "It may take some time to make that automatic but we are getting closer every day". Such tools would improve at least the `Upstream Sync` and `Dependency Management` activities. M1 mentioned two promising license analysis tools, while M3 remarked that "We already have automated testing tools in Ubuntu (see QA team) so we are heading in the right direction here". M6 saw the advent of atomic `Dependency Management` and other packaging process improvements as a promising development.

Overall, the six maintainers liked the work and found that the documented activities described their daily activities "quite well" (M6). They would not necessarily use our documented representation of the activities themselves (it is more targeted towards novices), except to systematically check which activities their distribution is not tracking (M4). Some missing important activities were identified, in particular license analysis and tracking of licensing changes, vulnerability resolution and post-release maintenance, as well as some missing challenges (especially time pressure). Finally, some tool support for dependency checking is expected to arrive in the medium term, however many challenges remain open.

7 Threats to validity

With respect to construct validity, there are several threats to consider. First, we used the change log messages as a representative record of the maintainers' activ-

ities, based on which important bug reports were identified for in-depth manual analysis and (if necessary) mailing list messages and other kinds of documentation. We did not formally verify the accuracy of these data sources, nor their completeness. Although M6 warned that the log message of the first version of a Debian package does not always mention whether `Local Patch` has been performed, none of the 4 instances of `New Package` found suffered from this issue.

There is no further evidence that suggests that the logs are incorrect: the three analyzed distributions require their maintainers to provide log messages [22, 46], since those are the primary input for end users and other maintainers affected by changes to a package. In fact, bug reports and mailing lists form the official means of communication in OSS distributions, together with IRC chat messages. In cases where a bug report identifier was missing (cf. Figure 4), either the change log item was sufficiently clear or we were able to find a related email message via a web search.

Second, we only analyzed a subset of the package-versions, and, hence, change logs. To mitigate this threat, we randomly sampled a large enough subset of package-versions to obtain a confidence interval of $\pm 5\%$ with a 95% confidence level. Furthermore, the activities that we identified for Ubuntu and FreeBSD did not add any new activity on top of those identified for Debian.

Third, our algorithm for reconstructing "versions" from the FreeBSD CVS commits depends on conventions that are documented by the FreeBSD project, but not explicitly enforced. It is possible that the recovered versions are either too fine-grained (under-approximating the actual number of activities performed for a version) or too coarse-grained (over-approximating). Feedback from the package maintainers confirmed that the algorithm is correct and that deviations from the guidelines should be minimal.

Fourth, since we study individual package-versions, our sample could contain multiple versions of some packages, just one version of other packages, and no version at all of the remaining packages. Such an approach is necessary, since large projects like KDE or GNOME involve more integration effort than smaller projects, and hence need to have more weight in our study. In addition, such projects typically also have a larger number of associated packages, which increases their weight further. The risk that this sampling decision biases the observed activities is small, since ecosystems like KDE and GNOME consist of hundreds of different applications and tools, developed by hundreds of developers and packaged by dozens of maintainers. In other words, even inside one such ecosystem, we should still expect a large diversity in integration activities.

Regarding internal validity, as mentioned above we rely on the accuracy and completeness of the logs of each package-version. Even in the event that some activities were not documented in the logs, there is no specific reason to believe that some activities would be less documented than others, hence this effect would cancel itself out across the different activities. For example, `Post-release Maintenance` was missed in our results, since "unimportant" discussions (i.e., those without explicit bug report or patch attached to them) did not have any trace in the change log and its referenced bug reports, across all three distributions.

Furthermore, the nature of manual classification implies that there might be some misclassifications (both for the activities as well as challenges). To overcome

this, the logs were interpreted by two of the authors, both of whom have experience in integration tasks (one of them is a Debian/Kubuntu developer), and they discussed their decisions with each other in order to resolve differences and obtain consensus. These discussions also resolved possible bias introduced by having the first set of tags be derived only by one of the authors. Furthermore, to validate the discovered patterns of integration and open challenges, we reached out to six maintainers/release engineers of Debian, Ubuntu and FreeBSD to evaluate and provide feedback on these patterns. Nonetheless, the quantitative results of this paper (prevalence of each activity) is exploratory only and we do not extrapolate these results.

The evaluation by the six maintainers was performed entirely via email, since this is the preferred means of communication for maintainers (and bug repositories, as discussed, are not suited). Furthermore, the asynchronous nature of emails provided breathing space to the maintainers and made it easier for them to organize their feedback amongst their voluntary open source activities and day-time job. Even then, we still observed that some of the questions were not addressed. In future work, we might complement asynchronous messages via email with synchronous follow-up via, for example, instant messaging (using IRC).

The open replies by some of the maintainers, as well as the selection of maintainers for the evaluation, also could introduce bias. M2 provided three open replies, M5 two open replies and M1 one open reply, yielding a total of 6 open replies out of 30 (20%). Due to the distribution of the open replies across the questions, each question obtained at least four concrete replies (two obtained six replies). Furthermore, the open replies are spread across the Debian and Ubuntu maintainers, reducing the overal impact of the missing data even further. Regarding selection bias, all six maintainers were experienced maintainers in their respective OSS distribution, covering a range of different packages according to size and domain.

An alternative evaluation methodology would have been to first perform a survey or interview, after which the research findings would be empirically analyzed and validated on change log and other data. However, doing this would bias our results to the activities that stakeholders think would be important, not necessarily *all* important activities that they are actually doing. Some essential activities never would have surfaced.

With respect to external validity, we have analyzed three of the largest OSS distributions as exemplars of packaging organizations. Since integration is the central activity of OSS distributions, we expect the identified activities to be representative for many of the activities that other packaging organizations would face in the case of OSS reuse. For example, packaging organizations like GNOME and KDE, or even "regular" Java or C++ systems that reuse multiple open source libraries as well have to deal with `Upstream Sync` (e.g., reusing a new version of log4j), `Dependency Management` (e.g., adding the dependencies of the new version of log4j) and `Local Patch` (e.g., customizing the new version of log4j to fix a bug). Nevertheless, manual analysis of other kinds of OSS distributions (e.g., Fedora-based), packaging organizations in general or any organization that performs multi-component integration, is necessary to confirm these conjectures and validate the generalizability of the seven integration activities. Such an analysis

might discover new activities, for example in the case of package organizations that do not build products for end-users but rather middleware or frameworks for other companies to build on.

8 Conclusion

Software reuse is a major tenet of software engineering, yet the integration activities that accompany it, be it in a COTS, OSS or ISS context, introduce unforeseen maintenance costs. Since more empirical research is necessary in this area to help organizations reuse components successfully and since most studies thus far focused on integration of individual components and/or non-OSS integration, we performed a large-scale study on three successful OSS distributions, i.e., Debian, Ubuntu and FreeBSD.

Analysis of a large sample of change log messages, bug reports and other historical integration data resulted in the identification of seven major integration activities, whose processes were documented in a pattern-like fashion to help organizations and researchers understand the responsibilities involved in integration. The activities were shown to be non-trivial and requiring a large amount of effort, and they were validated by six maintainers of the three distributions. Based on the seven documented activities, the major challenges turned out to be related to cherry-picking of safe changes from a new upstream release, the management of dependencies between packages, testing of packages and co-ordination among maintainers. Models and tools are needed to support these integration activities.

By providing a unified terminology across distributions and by documenting the integration activities in a structured way, our catalogue of activities enables maintainers of open source distributions, organizations interested in reusing OSS or ISS components, and researchers to better understand the challenges and activities that they face, and to plan policies, tools and methods to address these challenges. Together with other studies on integration, a dedicated training program on integration could be built, aimed at developers and their managers, with the aim of reducing or at least stabilizing maintenance costs caused by integration.

Finally, and very encouragingly, all distribution maintainers that we contacted hope that the documented activities and challenges will inspire researchers to start up a research program in the domain of reuse and integration.

Acknowledgments

The authors would like to thank all maintainers and release engineers of Debian, Ubuntu and FreeBSD who participated in our study, either directly (providing feedback on the documented activities), or indirectly (providing insights into the fascinating world of OSS distributions).

References

1. Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. Design recovery and maintenance of build systems. In *Proc. of the Intl. Conf. on Soft. Maint.*, pages 114–123, 2007.
2. Bram Adams, Ryan Kavanagh, Ahmed E. Hassan, and Daniel M. German. Replication package. http://mcis.polymtl.ca/publications/2015/integration_oss_distribution_adams_et_al.zip, 2011.
3. Christian Bac, Olivier Berger, Véronique Deborde, and Benoit Hamet. Why and how-to contribute to libre software when you integrate them into an in-house application ? *Proc. of the 1st Intl. Conf. on Open Source Systems (OSS)*, pages 113–118, June 2005.
4. Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. How reuse influences productivity in object-oriented systems. *Commun. ACM*, 39(10):104–116, October 1996.
5. Andrew Begel, Nachiappan Nagappan, Christopher Poile, and Lucas Layman. Coordination in large-scale software teams. In *Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, CHASE '09, pages 1–7, Washington, DC, USA, 2009. IEEE Computer Society.
6. Jesal Bhuta, Chris Mattmann, Nenad Medvidovic, and Barry W. Boehm. A Framework for the Assessment and Selection of Software Components and Connectors in COTS-Based Architectures. In *WICSA*, page 6, 2007.
7. Information Technology Resources Board. Assessing the risks of commercial-off-the shelf applications. Technical report, ITRB, September 1999.
8. Barry Boehm and Chris Abts. Cots integration: Plug and pray? *Computer*, 32(1):135–138, January 1999.
9. Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a case study: its extracted software architecture. In *Proc. of the 21st Intl. Conf. on Software Engineering (ICSE)*, pages 555–563, 1999.
10. Frederick P. Brooks, Jr. *The Mythical Man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
11. Lisa Brownsword, Tricia Oberndorf, and Carol A. Sledge. Developing new processes for cots-based systems. *IEEE Softw.*, 17(4):48–55, July 2000.
12. Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *Proc. of the 19th ACM SIGSOFT Symp. and the 13th European Conf. on Foundations of Software Engineering (ESEC/FSE)*, pages 168–178, 2011.
13. Weibing Chen, Jingyue Li, Jianqiang Ma, Reidar Conradi, Junzhong Ji, and Chunnian Liu. An empirical study on software development with open source components in the chinese software industry. *Softw. Process*, 13:89–100, January 2008.
14. W. G. Cochran. *Sampling Techniques*. John Wiley and Sons, Inc., New York, 2nd edition, 1963.
15. James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Softw.*, 15:37–45, Nov. 1998.
16. Ivica Crnkovic and Magnus Larssom. Challenges of component-based development. *J. Syst. Softw.*, 61(3):201–212, April 2002.
17. Bill Curtis, Herb Krasner, and Neil Iscoe. A field study of the software design process for large systems. *Commun. ACM*, 31(11):1268–1287, November 1988.
18. Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. In *Proc. of the 30th Intl. Conf. on Software Engineering (ICSE)*, pages 481–490, 2008.
19. Cleidson R. B. de Souza, David Redmiles, Li-Te Cheng, David Millen, and John Patterson. Sometimes you need to see through walls: A field study of application programming interfaces. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, CSCW '04, pages 63–71, New York, NY, USA, 2004. ACM.
20. Cleidson R. B. de Souza and David F. Redmiles. An empirical study of software developers' management of dependencies and changes. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 241–250, New York, NY, USA, 2008. ACM.
21. Project participants. http://www.debian.org/devel/people, 2013.
22. Debian project. *Debian Developer's Reference*, 2011 edition, 2011.
23. Robert DeLine. Avoiding packaging mismatch with flexible packaging. In *Proc. of the 21st Intl. Conf. on Software Engineering (ICSE)*, pages 97–106, 1999.

24. Developer's Reference Team, Andreas Barth, Adam Di Carlo, Raphaël Hertzog, Lucas Nussbaum, Christian Schwarz, and Ian Jackson. *Debian*. The Debian Project, April 2011.
25. Roberto Di Cosmo, Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio, and Stefano Zacchiroli. Supporting software evolution in component-based foss systems. *Sci. Comput. Program.*, 76:1144–1160, December 2011.
26. Piergiorgio Di Giacomo. Cots and open source software components: are they really different on the battlefield? In *Proc. of the 4th intl. conf. on COTS-Based Software Systems (ICCBSS)*, pages 301–310, 2005.
27. M. Dogguy, S. Glondu, S. Le Gall, and S. Zacchiroli. Enforcing type-safe linking using inter-package relationships. In *Proc. of the 21st Journées Francophones des Langages Applicatifs (JFLA)*, page 25p., January 2010.
28. William Frakes and Carol Terry. Software reuse: metrics and models. *ACM Comput. Surv.*, 28(2):415–435, June 1996.
29. William B. Frakes and Kyo Kang. Software reuse research: Status and future. *IEEE Trans. Softw. Eng.*, 31:529–536, July 2005.
30. Freebsd porter's handbook. http://bit.ly/FQDPhP, 2011.
31. The freebsd developers. http://www.freebsd.org/doc/en/articles/ contributors/staff-committers.html, 2013.
32. J. E. Gaffney and T. A. Durek. Software reuse – key to enhanced productivity: some quantitative models. *Inf. Softw. Technol.*, 31(5):258–267, June 1989.
33. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.
34. Daniel M. German, Jesus M. Gonzalez-Barahona, and Gregorio Robles. A model to understand the building and running inter-dependencies of software. In *Proc. of the 14th Working Conf. on Reverse Engineering (WCRE)*, pages 140–149, 2007.
35. Daniel M. German and Ahmed E. Hassan. License integration patterns: Addressing license mismatches in component-based development. In *Proc. of ICSE*, pages 188–198, 2009.
36. Daniel M. German, Jens H. Webber, and Massimiliano Di Penta. Lawful software engineering. In *Proc. of the FSE/SDP wrksh. on Future of Soft. Eng. research (FoSER)*, pages 129–132, 2010.
37. Jesus M. Gonzalez-Barahona, Gregorio Robles, Martin Michlmayr, Juan José Amor, and Daniel M. German. Macro-level software evolution: a case study of a large software compilation. *Empirical Softw. Engg.*, 14:262–285, June 2009.
38. Sigi Goode. Something for nothing: management rejection of open source software in australia's top firms. *Inf. Manage.*, 42(5):669–681, July 2005.
39. The BSD Certification Group. Bsd usage survey. Technical report, The BSD Certification Group, October 2005.
40. Øyvind Hauge, Claudia Ayala, and Reidar Conradi. Adoption of open source software in software-intensive organizations - a systematic literature review. *Inf. Softw. Technol.*, 52(11):1133–1154, November 2010.
41. Øyvind Hauge, Carl-Fredrik Sørensen, and Reidar Conradi. Adoption of open source in the software industry. In *Proc. of the 4th IFIP WG 2.13 Intl. Conf. on Open Source Systems (OSS)*, volume 275, pages 211–221, 2008.
42. James D. Herbsleb and Rebecca E. Grinter. Splitting the organization and integrating the code: Conway's law revisited. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 85–95, New York, NY, USA, 1999. ACM.
43. James D. Herbsleb, Audris Mockus, Thomas A. Finholt, and Rebecca E. Grinter. An empirical study of global software development: Distance and speed. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 81–90, Washington, DC, USA, 2001. IEEE Computer Society.
44. Raphaël Hertzog. Towards debian rolling: my own debian CUT manifesto. http://raphaelhertzog.com/2011/04/27/towards-debian-rolling-my-own-debian-cut-manifesto/, 2011.
45. Ari Jaaksi. Experiences on product development with open source software. In *Proc. of the IFIP Working Group 2.13 on Open Source Soft*, volume 234, pages 85–96. Springer, 2007.
46. Joseph Koshy. Building Products with FreeBSD. http://www.freebsd.org/doc/en/articles/building-products/, 2013. May 2013.

47. Foutse Khomh, Tejinder Dhaliwal, Ying Zou, and Bram Adams. Do faster releases improve software quality? – an empirical case study of mozilla firefox. In *Proc. of the 9th IEEE Working Conf. on Mining Software Repositories (MSR)*, pages 179–188, Zurich, Switzerland, June 2012.
48. Robert Lemos. Microsoft details new security plan. http://news.cnet.com/Microsoft-details-new-security-plan/2100-1002_3-5088846.html, 2003.
49. Patrick Lewis, Patrick Hyle, Marian Parrington, Elizabeth Clark, Barry Boehm, Christopher Abts, and Robert Manners. Lessons learned in developing commercial off-the-shelf (cots) intensive software systems. Technical report, SERC, October 2000.
50. Jingyue Li, Reidar Conradi, Christian Bunse, Marco Torchiano, Odd Petter N. Slyngstad, and Maurizio Morisio. Development with off-the-shelf components: 10 facts. *IEEE Softw.*, 26:80–87, March 2009.
51. Jingyue Li, Reidar Conradi, Odd Petter Slyngstad, Marco Torchiano, Maurizio Morisio, and Christian Bunse. A state-of-the-practice survey of risk management in development with off-the-shelf software components. *IEEE Trans. Softw. Eng.*, 34:271–286, 2008.
52. Jingyue Li, Reidar Conradi, Odd Petter N. Slyngstad, Christian Bunse, Umair Khan, Marco Torchiano, and Maurizio Morisio. An empirical study on off-the-shelf component usage in industrial projects. In *Proc. of the 6th intl. conf. on Product Focused Software Process Improvement (PROFES)*, pages 54–68, 2005.
53. Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag, 2007.
54. Frank Van Der Linden. Applying open source software principles in product lines. *The European journal for the informatics professional (UPGRADE)*, 3:32–40, June 2009.
55. Andreas Lundqvist. Gnu/linux distribution timeline. http://futurist.se/gldt/, 2013.
56. Michael Mattsson, Jan Bosch, and Mohamed E. Fayad. Framework integration problems, causes, solutions. *Commun. ACM*, 42(10):80–87, October 1999.
57. Stephen McCamant and Michael D. Ernst. Predicting problems caused by component upgrades. In *Proc. of the Symposium on the Foundations of Software Engineering*, pages 287–296, 2003.
58. Shane McIntosh, Bram Adams, Yasutaka Kamei, Thanh Nguyen, and Ahmed E. Hassan. An empirical study of build maintenance effort. In *Proc. of ICSE*, pages 141–150, May 2011.
59. Janne Merilinna and Mari Matinlassi. State of the art and practice of opensource component integration. In *Proc. of the 32nd Conf. on Software Engineering and Advanced Applications (EUROMICRO)*, pages 170–177, 2006.
60. Marc H. Meyer and Alvin P. Lehnerd. *The Power of Product Platforms*. Free Press, March 1997.
61. Martin Michlmayr, Francis Hunt, and David Probert. Release management in free software projects: Practices and problems. In *Open Source Development, Adoption and Innovation*, volume 234, pages 295–300, 2007.
62. Ivan Mistrík, John Grundy, André Hoek, and Jim Whitehead. *Collaborative Software Engineering: Challenges and Prospects*, chapter 19, pages 389–402. Springer, first edition, 2010.
63. Abdallah Mohamed, Guenther Ruhe, and Armin Eberlein. Optimized mismatch resolution for cots selection. *Softw. Process*, 13(2):157–169, March 2008.
64. M. Morisio, C. B. Seaman, V. R. Basili, A. T. Parra, S. E. Kraft, and S. E. Condon. Cots-based software development: processes and open issues. *J. Syst. Softw.*, 61(3):189–189, April 2002.
65. Fredy Navarrete, Pere Botella, and Xavier Franch. How agile cots selection methods are (and can be)? In *Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, EUROMICRO '05, pages 160–167, Washington, DC, USA, 2005. IEEE Computer Society.
66. Heikki Orsila, Jaco Geldenhuys, Anna Ruokonen, and Imed Hammouda. Update propagation practices in highly reusable open source components. In *Proc. of the 4th IFIP WG 2.13 Int. Conf. on Open Source Systems (OSS)*, volume 275, pages 159–170, 2008.
67. D. L. Parnas. On the design and development of program families. *IEEE Trans. Softw. Eng.*, 2:1–9, Jan. 1976.
68. Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., 2005.
69. Scott James Remnant. A new release process for ubuntu? http://netsplit.com/2011/09/08/new-ubuntu-release-process/, 2011.

70. Josip Rodin and Osamu Aoki. *Debian New Maintainers' Guide*. The Debian Project, June 2011.
71. Michel Ruffin and Christof Ebert. Using open source software in product development: A primer. *IEEE Softw.*, 21(1):82–86, January 2004.
72. Bert M. Sadowski, Gaby Sadowski-Rasters, and Geert Duysters. Transition of governance in a mature open software source community: Evidence from the debian case. *Information Economics and Policy*, 20(4):323–332, 2008.
73. Walt Scacchi, Joseph Feller, Brian Fitzgerald, Scott Hissam, and Karim Lakhani. Understanding free/open source software development processes. *Software Process: Improvement and Practice*, 11(2), 2006.
74. Carolyn B. Seaman. Communication costs in code and design reviews: An empirical study. In *Proceedings of the 1996 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '96, pages 34–. IBM Press, 1996.
75. Emad Shihab, Christian Bird, and Thomas Zimmermann. The effect of branching strategies on software quality. In *Proc. of the ACM/IEEE intl. symp. on Empirical Software Engineering and Measurement (ESEM)*, pages 301–310, 2012.
76. Mark Shuttleworth. The art of release. http://www.markshuttleworth.com/archives/146, 2008.
77. Manuel Sojer and Joachim Henkel. Code Reuse in Open Source Software Development: Quantitative Evidence, Drivers, and Impediments. *Journal of the Association for Information Systems*, 11(iss.12), 2010.
78. Diomidis Spinellis and Clemens Szyperski. Guest editors' introduction: How is open source affecting software development? *IEEE Softw.*, 21(1):28–33, January 2004.
79. Klaas-Jan Stol, Muhammad Ali Babar, Paris Avgeriou, and Brian Fitzgerald. A comparative study of challenges in integrating open source software and inner source software. *Inf. Softw. Technol.*, 53(12):1319–1336, December 2011.
80. Clemens Szyperski. *Component software: beyond object-oriented programming*. Addison-Wesley Publishing Co., 1998.
81. The Fedora Project. Package update HOWTO. http://fedoraproject.org/wiki/Package_update_HOWTO, 2011.
82. The FreeBSD Documentation Project. *FreeBSD Porter's Handbook*. The FreeBSD Foundation, 2011.
83. F. Tiangco, A. Stockwell, J. Sapsford, A. Rainer, and E. Swanton. Open-source software in an occupational health application: the case of heales medical ltd. *Procs*, 1:130–134, 2005.
84. Paulo Trezentos, Inês Lynce, and Arlindo L. Oliveira. Apt-pbo: solving the software dependency problem using pseudo-boolean optimization. In *Proc. of the IEEE/ACM intl. conf. on Automated Software Engineering (ASE)*, pages 427–436, 2010.
85. Qiang Tu and Michael Godfrey. The build-time software architecture view. In *Proc. of ICSM*, pages 398–, 2001.
86. "motu" team. https://launchpad.net/%7Emotu/+members, 2013.
87. "ubuntu core development team" team. https://launchpad.net/%7Eubuntu-core-dev/+members, 2013.
88. "ubuntu universe contributors" team. https://launchpad.net/ universe-contributors/+members, 2013.
89. André van der Hoek and Alexander L. Wolf. Software release management for component-based software. *Softw. Pract. Exper.*, 33:77–98, January 2003.
90. Kris Ven and Herwig Mannaert. Challenges and strategies in the use of open source software by independent software vendors. *Inf. Softw. Technol.*, 50(9-10):991–1002, August 2008.
91. James Whittaker, Jason Arbon, and Jeff Carollo. *How Google Tests Software*. Addison-Wesley Professional, 2012.
92. Comparison of bsd operating systems. http://en.wikipedia.org/wiki/Comparison_of_BSD_operating_systems, 2011.
93. Xin Xia, David Lo, Feng Zhu, Xinyu Wang, and Bo Zhou. Software internationalization and localization: An industrial experience. In *Proc. of the 18th Intl. Conf. on Engineering of Complex Computer Systems (ICECCS)*, pages 222–231, 2013.
94. Daniil Yakimovich, James M. Bieman, and Victor R. Basili. Software architecture classification for estimating the cost of cots integration. In *Proc. of the 21st Intl. Conf. on Software Engineering (ICSE)*, pages 296–302, 1999.