Sampling in Open Source Software Development: The case for using the Debian GNU/Linux Distribution

WORKING DRAFT 2006-06-16

Sebastian Spaeth, Matthias Stuermer,
Stefan Haefliger, Georg von Krogh, ETH Zurich

Abstract*

Research on open source (OS) projects often focuses on the SourceForge collaboration platform. We argue that a GNU/Linux distribution, such as Debian, is better suited for the sampling of projects because it avoids biases and contains unique information only available in an integrated environment. Especially research on the reuse of components can build on dependency information inherent in the Debian GNU/Linux package tracking system.

1. Introduction

One of the central problems of research design for OS projects is sampling. Hundreds of thousands of OS projects exist, some of which are not registered on any collaboration platform or do not have a corresponding website. Many researchers turn to SourceForge.net, using projects registered there as population (e.g. Crowston and Howison, 2004; Madey et al., 2002). SourceForge is arguably the largest collaboration platform: it currently hosts 122,000 projects and has more than 1.3m users registered. Despite its size, sampling from SourceForge means systematically omitting projects hosted on other platforms (such as Savannah, Tigris, or Berlios) and those not listed on such platforms at all. SourceForge was intended specifically as an incubator for small projects, therefore potentially biasing the sample further. Large projects, such as Apache, Samba, Mozilla, or the Linux kernel are able to host their own infrastructure and are not registered on SourceForge. Others use only part of the offered tools, such as the file distribution or the CVS repository, and ignore for instance the included bug database or dump old and unmaintained data in their code repository. Howison and Crowston (2004) have recognized this in an article which explicitly warns from blindly mining SourceForge project data.

When examining open source projects, we propose to look at software distributions containing entire populations of OS projects; especially Debian seems an appropriate target.

"The Debian Project[1]" is one of the oldest GNU/Linux distributions in existence, founded by Ian Murdock on Aug 16th, 1993 (Debian.org, 2006). It builds on the GNU operating system and the Linux kernel. Debian is maintained by a large community of individuals, hence one of the few large distributions without a commercial background. Many other GNU/Linux distributions, for instance Ubuntu, are directly derived from it. Debian offers a vast package repository compared to other distributions, and most OS applications, license permitting, can be found there. Debian's history and comprehensiveness make it attractive to study as it represents a large universe of OS projects.

We argue that sampling projects from Debian overcomes some of the major sampling problems incurred when using single collaboration platforms, as Debian includes software components hosted on various platforms, and even those without a website of their own.

The work of integrating thousands of such packages into a distribution results in a repository of software components that can be navigated along various criteria. One of the interesting attributes to research is the dependency information, allowing us to study the amount of reuse for each component.

The study of competition among OS projects has been delayed by the difficulty of measuring (relative) success. Relational information among OS projects, as it is available from an integrated environment of software components, enables an analysis of reuse

* The authors wish to thank Christian Sommer, Daniel Baumann and Mario Bischof for their valuable research assistance and two Debian developers for their comments.

1 The official pronunciation of Debian is 'deb ee n'. The name originates from the names of the creator of Debian, Ian Murdock, and his wife, Debra.

across projects. Every instance of component reuse saves overall development cost. Therefore, frequent reuse, as we argue, amounts to relative success of a program and invites empirical studies that aim at predicting reuse success. In this paper, we discuss the sampling that precedes a study of competitive dynamics among OS projects and propose the Debian distribution as a suitable environment for such a sampling.

This paper is organized as follows. The next section provides the outline of an ongoing research project on software reuse in order to illustrate the use of a distribution as an empirical base when analyzing competitive dynamics among OS software components. The third section reviews the critical sampling issues when using the Debian distribution and discusses data collection in more detail. The last section concludes with some limitations and implications for OS research.

2. The case of software reuse

The development of open source software reveals new innovation practices that promise fruitful insights to students of organizations (von Krogh and von Hippel, 2006). While the study of innovation processes in open source software development has received considerable attention with regards to the developers' motivations (Hertel et al., 2003; Lakhani and Wolf, 2005), project governance (Lee and Cole, 2003; O'Mahony, 2003; Shah, 2006), and coordination (Bonaccorsi and Rossi, 2003), the study of competition among open source software products has received little attention due to the difficulty to measure competitive performance satisfactorily where monetary rewards and gains are virtually absent (Crowston et al., 2003). An analysis of software reuse can both serve to discriminate between more or less successful open source software projects and hold implications for firms and the management of innovation.

One of the central problems in the management of innovation is if and how firms reuse previously created knowledge across the various stages of an innovation process (see Argote, 1999; Majchrzak, Cooper, and Neece, 2004; von Krogh, Ichijo, and Nonaka, 2000; Zander and Kogut, 1995). Research on reuse of software shows that implementing reuse is difficult for organizations to achieve and depends on many organizational settings which explicitly encourage programmers to conduct reuse (Lynex and Layzell, 1998; Banker and Kauffman, 1991). Also, the creation of easily reusable components needs explicit encouragement as the creation of reusable software requires an additional planning and implementation effort in areas such as documentation, portability, architectural design and collaboration. Literature on software reuse estimates additional costs of up to 200 percent on top of the production costs for making software components reusable (Tracz, 1995).

A recent study among open source software development projects revealed that reuse is widespread and frequent (von Krogh et al., 2006). Understanding the incentives at work in open source software development helps to explain the extent of code reuse in general. We argue that an integrated environment of software components can inform information systems research and, ultimately, strategy research about the success factors of components in terms of reuse frequency. By studing the given information in Debian and coding additional characteristics of software components, we ask: what are the characteristics of software components that are reused more often than others? And how can we predict reuse success?

Open source software components come in a myriad of forms and originate from diverse contexts. Ultimately, however, some are reused more frequently than others (see Figure 1). Information Systems (IS) literature, and research on software reuse in particular, predicts that developers will reuse components that help lowering the development costs for the firm or the individual (Banker et al., 1993; Prieto-Diaz, 1993; Frakes and Isoda, 1994; Griss, 1993; von Krogh et al., 2006). In open source software development, considerable reuse of components is conducted despite the lack of explicit organizational encouragement and a plethora of ready-to-use components are publicly available for free in the pool of open source projects (von Krogh et al., 2006). Software developers reuse if their development costs can be mitigated through reuse relative to writing the software from scratch (Banker et al., 1993). Standards and tools for classification and retrieval help to lower the costs for search and integration of a component (Isakowitz and Kauffman, 1996; Poulin, 1995). Hence, the organizational implications for component developers are as follows: Published, easily identifiable, and well documented components should facilitate reuse.

In addition to searching a component, the developer needs to trust and then integrate a component into his or her system in order to reuse it. Information accompanying reusable software should contain quality ratings or certification to enhance the developer's trust in a component written by someone else (Knight and Dunn, 1998; Poulin, 1995). Taking into consideration the requirements for successful reuse, a component should lower the prospective reuser's search and integration costs and communicate confidence in component's quality.

As already elaborated, the Debian distribution offers an ideal context for studying the variance of reuse across software components. A comprehensive tracking system structures the packages managed and distributed

by Debian[2]. These packages are created by the Debian developers acting as maintainers who bundle a so-called upstream project – the piece of software intended to be integrated into the Debian distribution – with various scripts, e.g. for the configuration and installation of the upstream software into the particular Debian system. Additionally, the Debian maintainer adds meta-information about the package, like a short description of the component's functionality or the software section the package belongs to. Most important, the maintainer logs the dependencies of this package, that is, which other packages are needed to be installed first in order to execute the intended software. Therefore, each package comes with dependency information, leading to a dependency graph: If a user of the distribution wishes to run program x, the dependency information can be used to check if all other packages needed by program x are installed or need to be installed on the user's machine. The reverse dependency, that is which programs depend on package y to function properly, can be understood as instances of reuse.

An abundance of other package information can serve as potential independent variables for a model. They include:

- Number of binary packages per source package.
- Size of the source package.
- Size of the binary packages.
- Debian bug statistics: Amount and urgency rating.
- Age of the Debian package.
- Change logs for the package within Debian.
- License information.
- Identity of maintainers and, usually, authors of the package.
- External (upstream) source of the package.
- Version information.

Depending on the focus of the reuse model and the required variables, additional project information for each Debian package should be collected from the communities that program and maintain the software in order to properly test that model. Researchers should avoid fitting their theories to the available data (Howison and Crowston, 2004).

Studying dependencies among Debian packages could shed light on the characteristics of components that are more frequently reused. As each incidence of reuse saves overall coding costs, the variance in reuse can be interpreted as success among developers of other software components and hence as competitive success with respect to similar components.

2 The Debian Package Tracking System can be found here: http://packages.qa.debian.org/common/

3. Sampling and data collection

For research questions that probe into the comparative or competitive dynamics of OS projects, a complete, unbiased, and high-variance sample is difficult to obtain. Particularly for large-scale populations, researchers often rely on collaboration platforms such as SourceForge with its known pitfalls (Howison and Crowston, 2004). Research projects that require reliable, comprehensive, standardized, and relational information across OS software projects may find it advantageous to sample projects from Debian. Some of the advantages include the following.

- Debian developers assemble a comprehensive universe of software products.
- Debian contains programs which are actually used, as at least one person insisted it be included in the repository.
- Rules and guidelines ensure a standardized process of including software into the distribution[3]. As a result, the information available for each package is usually complete (information listed above) and dates back until up to 1993.
- Packages within one version of the distribution are designed to be compatible, hence form an integrated environment of software products.
- Categories, or sections, within the distribution provide possible sampling foci (sections include editors, web, admin, libs, mail, and so on)
- For each package, a maintainer assumes personal responsibility and is knowledgeable about the software he or she maintains for Debian.

The current stable Debian release, termed 'sarge', was examined, including security updates as of May 2006. All 'main' and 'contrib' (3rd party contribution) packages were used, only excluding 'unfree' packages[4] which contain popular, yet non-free software, such as Macromedia's Flash or the Realtime player. This led to a repository of 19,692 binary packages for 32bit Intel-compatible processors (i386), which were associated with a section attribute (such as mail, text, or libs). A binary package is the ready-to-run compiled version of one program or part of a program. Each binary package

3 The developer manual and reference can be found here: http://www.us.debian.org/doc/manuals/developers-reference/

4 for further information see the Debian Free Software Guidelines on http://www.debian.org/social_contract.en.html#guidelines

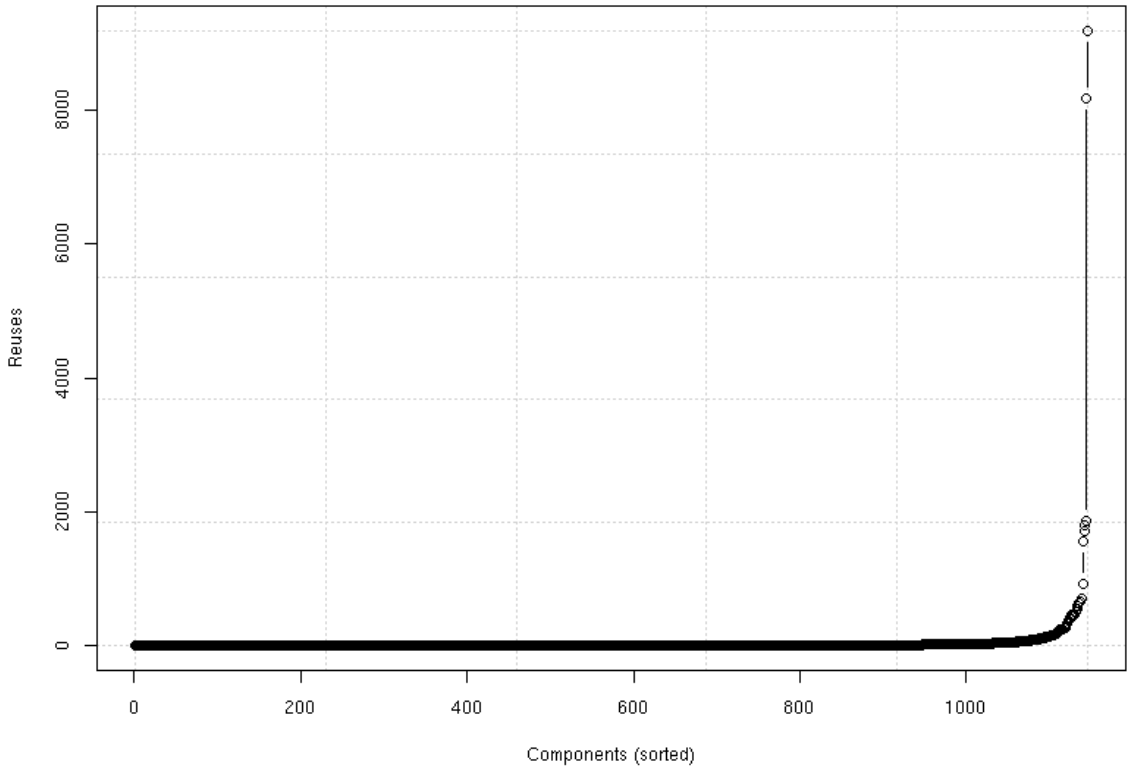

Figure 1: Component reuse(n=1146)

is compiled from exactly one corresponding source package, although one source package can be split up into several binary packages in Debian (these often offer additional but optional functionality), leading to a 1:n relationship (see schematics in Figure 2). An example for multiple binary packages derived from one source package "php3" is the binary core package "php3", plus the optional binary package "php3-xml" providing an xml interface to the PHP programming language. Discussions with two Debian developers confirmed that source packages represent the software component/project better than binary packages and should, hence, become the units of analysis. Of the 8,890 source packages, 1,146 source packages contained packages marked as "libs" or "oldlibs", meaning they were categorized by Debian developers as reusable library component.

The Debian repository archive contains information on the name, version, maintainer, license, size and other information about each package. Even more important, each package contains references to other packages which are required in order to run. In the case of reusable components this 'dependency information' is able to reveal how many programs depend on the component, that is how often the component is being (re-)used. Debian uses three categories of dependency information: 1) required, 2) recommended and 3) suggested. Since only dependencies which are *required* by another package are really needed to run that application, we termed this type of reuse a *hard reuse*. This type of reuse can seldom be influenced by the Debian maintainer of the project, but is exogenously decided by the "upstream" project. *Recommended* dependencies consist of programs which are useful to have or provide, say, the documentation for the application. *Suggested* dependencies include various packages that might be remotely useful for a given package. In contrast to hard reuses, the choice of recommended and suggested reuses can be defined by the Debian maintainer, and is far more ambiguous. In our definition, component reuse only refers to instances of hard reuse, which could be confirmed by Debian developers.

Dependencies, as reported in the package tracking system, can also specify alternatives: package x may require either package y or package z to be installed. In thees cases, we coded package x as reusing both package y and package z. The Debian bug tracker, which is used to enter and track errors in the software,

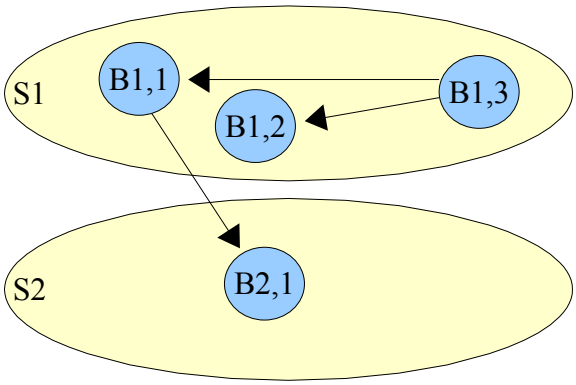

Figure 2: Binary & source packages

was used to extract the number of currently open bugs for each of the packages.

The 1,146 components were reused 48,054 times (not including reuses of binary packages that are derived from the same source package, or in other words, intra-component reuse). The reuse frequency for each component is visualized in Figure 1 and clearly follows a power law distribution. Reuse frequency ranges between 0 and 9,190. The first quartile is 0, the median 2, the third quartile 7. In average each component was reused 42.9 times.

Examining the reuse frequency of components, two projects proved to be statistical outliers: xfree86 (needed by every graphical program) and gcc (which contains a number of libraries for several programming languages and is required for most packages).

4. Discussion

While single collaboration platforms such as SourceForge can provide much data, they should be used with caution: focusing on projects on one of them could lead to a potential bias and the inclusion of inconsistent data. As an example: only 58% of 157 randomly selected Debian library packages were listed on Freshmeat.org (a popular site announcing version and summary statistics for OS projects) and just 39% were hosted on a collaboration platform. Thus, the remaining projects would have been missed if the data had been obtained from a single collaboration platform only. A distribution, on the other hand, represents the population of projects in use. It seems reasonable to use packages of large distributions as a representative population of OS projects.

There are limits however. Using a software distribution also limits the choice of projects in some ways. By looking at a Linux distribution we had -by definition- excluded components which run exclusively on, of example, the Windows platform.

A second potential disadvantage to use Debian as a sampling base is the 'success bias'. Only projects which actually provide some functionality and are deemed useful and good enough are included in the distribution. This omits “stub packages”, like many on SourceForge (Krishnamurty, 2002), where one person registers a project, dumps some code and waits for others to finish the work. This bias might become a problem when studying project failure. However, the stable branch of Debian contains many orphaned packages, which are not currently maintained but, nonetheless, had been included by a maintainer.

Third, Debian is very strict when it comes to licenses and excludes projects which do not adhere to a Free license. Packages that are not purely based on OS software, such as Java-based packages, might be underrepresented. Finally, although Debian provides many attributes for each package, some attributes, such as the license, or the programming language, cannot be retrieved in a systematic and automated manner yet. There is an ongoing effort, however, called debtags, to label each package systematically with more information.

Critics might point to the putative limitation of the Debian maintainer's subjective categorization of components into software sections. While sometimes ambiguous, the categorization is performed by a 3rd party and not through self-categorization as it is usually performed in collaboration platforms.

The study of competitive dynamics among OS projects has only started. As part of a long-term agenda for research on OS software development, von Krogh and von Hippel (2003) suggested to advance our understanding of competition in a context of voluntary contributions. The prerequisites for comparing OS projects include access to standardized data on the software products, such as license, programming language, age, domain, and, maybe more importantly, access to relational information that connects software programs to each other. Most software products work as part of larger systems, hardware but also software such as networks, operating systems, and platforms. To understand which programs dominate over others of a similar type, and why, researchers need a data base that allows for comparison.

Recent work on competition between OS software and proprietary software uncovered strategies for the firm to survive in the face of OS competition (Casadesus-Masanell and Ghemawat, 2006). But direct competition need not be the prevailing option for strategy. In many cases, firms can integrate OS software into their hardware and software architecture

(Henkel, 2006) and in turn may find it useful to contribute back to the community (Harhoff et al., 2003). Studies of reuse, both in OS and among proprietary software vendors, could help to understand the processes that govern mutual sharing of knowledge and code. Information on reuse can be assembled piecemeal (von Krogh et al., 2006) or incorporated as a global feature of the sample data, as we show here.

5. References

Argote, L. 1999. Organizational learning: Creating, retaining, and transferring knowledge. Kluwer, Norwell, MA.

Banker, R.D., R.J. Kauffman. 1991. Reuse and productivity in integrated computer-aided software engineering: An empirical study. MIS Quarterly. 15(3) 375-401.

Banker, R.D., R.J. Kauffman, D. Zweig. 1993. Repository evaluation of software reuse. IEEE Transactionson Software Engineering. 19(4) 379-389.

Bonaccorsi A., C. Rossi. 2003. Why Open Source software can succeed. Research Policy. 32(7) 1243-1258.

Casadesus-Masanell, R, P. Ghemawat. 2006. Dynamic Mixed Duopoly: A Model Motivated by Linux vs. Windows. Management Science. Forthcoming.

Crowston, K., H. Annabi, J. Howison. 2003. Defining Open Source Software Project Success. Proc. of International Conference on Information Systems (ICIS 2003)

Crowston, K. and J. Howison. 2004. "The social structure of free and open source software development", Syracuse FLOSS research working paper. http://opensource.mit.edu/papers/crowstonhowison.pdf.

Debian.org, "A Brief History of Debian", 2006, www.debian.org/doc/manuals/project-history/

Frakes, W., S. Isoda. 1994. Success factors of systematic reuse. IEEE Software. 11(5) 15-19.

Griss, M.L. 1993. Software reuse: From library to factory. IBM Systems Journal. 32(4) 548-566.

Harhoff, D., Henkel, J., von Hippel, E. 2003. Profiting from voluntary information spillovers: How users benefit by freely revealing their innovations. Research Policy 32 1753-69.

Henkel, J. 2006. Selective revealing in open innovation processes: The case of embedded linux. Working paper. URL: http://www.tim.wi.tum.de/paper/Henkel_Selective_revealing_2005.pdf

Hertel, G., S. Niedner, S. Herrmann. 2003. Motivation of software developers in Open Source projects: an Internet-based survey of contributors to the Linux kernel. Research Policy. 32(7) 1159-1177.

J. Howison, K. Crowston. 2004. The perils and pitfalls of mining sourceforge". http://opensource.mit.edu/papers/howison04msr.pdf.

Isakowitz, T., R.J. Kauffman. 1996. Supporting search for reusable software objects. IEEE Transactions on Software Engineering. 22(6) 407-423.

Knight, J.C., M.F. Dunn. 1998. Software quality through domain-driven certification. Annals of Software Engineering. 5 293-315.

Krishnamurthy, S. 2002. Cave or community? An empirical examination of 100 mature open source projects. First Monday. 7(6).

Lakhani, K.R., R.G. Wolf. 2005. Why hackers do what they do: Understanding Motivation and effort in Free/Open Source software projects. In: Feller, J. Fitzgerald, B., Hissam S., K.R. Lakhani (eds.). Perspectives on Free and Open Source software. MIT Press.

Lee, G. K. and Cole, R. E. 2003. From a Firm-Based to a Community-Based Model of Knowledge Creation: The Case of the Linux Kernel Development. Organization Science, 14 (6), 633-649.

Lynex, A., P.J. Layzell. 1998. Organisational considerations for software reuse. Annals of Software Engineering. 5 105-124.

Madey, V. Freeh, R. Tynan, 2002. "The open source software development phenomen: An analysis based on social network theory", in: Americas Conference on Information Systems, Dallas, TX.

Majchrak, A., L.P. Cooper, O.P. Neece. 2004. Knowledge reuse for innovation. Management Science. 50(2) 174-188.

O'Mahony, S. 2003. Guarding the commons: How community-managed software projects protect their work. Research Policy. 32(7) 1179-1198.

Prieto-Diaz, R. 1993. Status report: Software reusability. IEEE Software. 10(3) 61-66.

Poulin, J.S. 1995. Populating software repositories: Incentives and domain-specific software. Journal of Systems Software. 30 187-199.

Ravichandran, T. 1999. Software reusability as synchronous innovation: A test of four theoretical models. European Journal of Information Systems. 8 183-199.

Shah, S. 2006. Motivation, governance, & the viability of hybrid forms in open source software development. Management Science. Forthcoming.

Tracz, W. 1995. Confessions of a used program salesman: Instiutionalizing software reuse. Addison-Wesley.

von Krogh, G., E. von Hippel. 2006. Introduction to the special issue. Management Science. Forthcoming.

von Krogh, G., K. Ichijo, I. Nonaka. 2000. Enabling Knowledge Creation: How to Unlock the Mystery of Tacit Knowledge and Release the Power of Innovation. Oxford University Press.

von Krogh, G., E. von Hippel. 2003. Special issue on open source software development. Editorial. Research Policy. 32(7) 1149-1157.

von Krogh, G., S. Haefliger, S. Spaeth. 2006. Knowledge sharing in open source software development: “Shifting the creative effort”. Best Paper Proceedings of the Academy of Management Conference, Atlanta.

Zander, U., B. Kogut. 1995. Knowledge and the speed of the transfer and imitation of organizational capability: An empirical test. Organization Science. 6(1) 76-92.