Using software distributions to understand the relationship among free and open source software projects

Daniel M. German
Software Engineering Group
Dept. of Computer Science
University of Victoria
dmg@cs.uvic.ca

Abstract

Success in the open source software world has been measured in terms of metrics such as number of downloads, number of commits, number of lines of code, number of participants, etc. These metrics tend to discriminate towards applications that are small and tend to evolve slowly. A problem is, however, how to identify applications in these latter categories that are important. Software distributions specify the dependencies needed to build and to run a given software application. We use this information to create a dependency graph of the applications contained in such a distribution. We explore the characteristics of this graph, and use it to define some metrics to quantify the dependencies (and dependents) of a given software application. We demonstrate that some applications that are invisible to the final user (such as libraries) are widely used by end-user applications. This graph can be used as a proxy to measure success of small, slowly evolving free and open source software.

1 Introduction

As the MSR series of workshops (and other conferences) demonstrate, free and open source software (FOSS) has become an important source of projects on which empirical studies are performed. In general most of these projects are considered independent entities, similar to the way software is developed in the proprietary world. But the FOSS ecology is rich in interaction among members of different software development teams, and it is reflected in the way that software is reused by other software. In [6] Spinellis and Szypersky cited how the Xine multimedia player required 11 different libraries. Xine developers, therefore, required to know what these libraries did, and changes in these libraries would have had an effect on Xine, making them stakeholders (and users) in their development. There is, unfortunately, very little research the highlights how FOSS is interrelated. Most of the work has been focused in understanding how the corresponding communities of FOSS developers interact between each other (for example [3]).

Software distributions play an important role in the open source software: they are the editors that pick software, package it, and offer it to people who are interested in using it. They simplify the task of building software by providing binaries to users that are readily available. RedHat, Debian, Fink, SUSE Linux, Fedora Core, are all FOSS distributions. Each distribution selects a set of software projects based on its goals and the needs of its users. Projects are usually referred as packages. Each of these packages is described usually in a file: the URI where the source of the package is located, its the license, a list of packages needed to build it and run it, and many other attributes. This file is then used by the package management system (PMS) of the distribution. Red Hat uses *yum*, Debian uses *pkg*, and Fink uses *fink* as PMS. The end-user interacts with the PMS, requesting a package to be installed (or deleted). The PMS resolves all dependencies, and pre-installs any other software required by the application to install. PMS have greatly simplified the management of software packages in the FOSS world. Robles et al [4] were the first to highlight software distributions as an important source of information regarding how software evolves. For their study they used the Debian distribution. They analyzed the evolution of the size of the distribution, the evolution in the size of the packages they include, and the distribution of programming languages used.

The main thesis of this paper is that information present in the list of dependencies of a package can be use to better understand how FOSS is related (i.e. what packages use what packages), and to help us identify packages that are important to the FOSS ecology. In this paper we formalize the dependencies among packages (according to a distribution), and define a set of metrics to quantify how important a

Fourth International Workshop on Mining Software Repositories (MSR'07)
0-7695-2950-X/07 $20.00 © 2007 IEEE

package is to other packages (and within a distribution). We also compare these metrics to those found in SourceForge.

For this paper we selected the distribution Fink[1]. Fink is a distribution of FOSS packages for OS X. It uses `.info` files to describe the metadata of its packages. Fink is widely used by Mac OS X users who are interested in installing FOSS software in a simple, yet effective manner.

2 Fink

We proceeded to download a snapshot of the collection of Fink `.info` files via its CVS repository (hosted at fink.sourceforge.net) on Dec. 2006. Figure 1 shows an excerpt of one of the packages. Fink maintains two sets of packages: stable and unstable. We wanted to avoid `.info` packages that might contain errors, and hence only used packages in stable. There were a total of 1207 `.info` files.

```
Package: aspell
Version: 0.50.5
Revision: 1002
Source: mirror:gnu:%n/%n-%v.tar.gz
Source-MD5: 14403d2ea5ded5d3fc9bb259bf65aab5
BuildDepends: libncurses5 (>= 5.4-200[...])
Depends: ncurses, libncurses5-shlibs
         (>= 5.4-20041023-1006), %n-shlibs
[...]
```

Figure 1. Excerpt of `.info` file.

Each info `.info` is composed of a list of fields[2]. Some some packages are offered in more than one variant (different versions of the same application, such as *perl*, *python*, or *postgreSQL*).

Each package is the smallest identifiable unit that can be independently installed using Fink. Packages are installed using the command `fink install <packagename>` or using a graphical user interface. Fink will download (if configured accordingly build) and install this package or any other package required to satisfy its dependencies.

It is important to distinguish the difference between package and software project. Each project might provide one or more packages. For example, the project *libpng*[3] (a library to read and write the PNG image format) provides the packages *libpng3* and *libpng-shlibs*.

We concentrated our attention in a subset of the fields in the `.info` file. These are listed in table 1. Each package-info file was parsed, and its fields loaded into a database. "Bundles" were split into each component package.

[1] `fink.sourceforge.net`

[2] In very few cases a package-info file is composed of a collection of embedded package descriptions itself (a "bundle").

[3] `libpng.sourceforge.net`

Field	Description
Package	Name of the package
Version	Version of the software project used
License	License used by the project
Depends	List of packages required to run the project
BuildDepends	List of packages required to build the project
Provides	An optional list of the packages provided
Source	URI where the source code is available

Table 1. Description of the most relevant fields in the info files

There were two fields in particular that required extra processing: *Depends* and *BuildDepends*. The former is a list of package names required to run the application, and the latter a list of packages needed to build the application. From the point of view of mining relationships between applications this separation is interesting as it has the potential to bring attention to applications that are used by a small set of users (the developers) yet have a huge impact to the final users (for example, compilers are hardly considered "popular" applications, yet their output is run by users over and over again).

The field *Provides* is a list of package names that this package is considered to "provide"[4]. We consider this list of names as "aliases" for the original package (more than one package might have the same alias; for example, *x11* is an alias for both *xfree86* and *xorg*).

We identified a total of 1217 packages.

Fink classifies `.info` files into a hierarchical structure. This structure is manifested using the file system and it is described in table 2. The table lists for each subdirectory the number of `.info` files placed in it.

The information was loaded into a relational database and several scripts were created to answer our research questions. Aisee[5] was used to render graphs.

3 Relationship among Packages

It is widely accepted that reuse is important in FOSS [7, 6, 2]. Given that most FOSS has no purchase cost associated with it[6] developers are likely to look for components (in the form of libraries, or stand-alone command-line pro-

[4] According to the Fink Reference Manual: "if a package named *pine* specifies Provides: *mailer*, then any dependency on *mailer* is considered satisfied when *pine* is installed." `http://fink.sourceforge.net/doc/packaging/reference.php?phpLang=en`.

[5] `www.aisee.com`

[6] There is a cost in terms of learning to use the software.

Directory	Count	Directory	Count
crypto/finkinfo	61	libs/rubymods	4
base	26	net	59
database	16	sci	86
devel	66	shells	7
editors	48	sound	32
games	74	text	98
gnome	121	utils	79
graphics	71	web	16
kde	18	x11	45
languages	50	x11-system	2
libs	134	x11-wm	10
libs/perlmods	84		

Table 2. Organization of `.info` files.

grams) that address part of the problem that they are trying to solve. It is therefore likely that a software package will use other software packages. The process of building a software package also requires a set of software packages (such as a compiler).

Using Fink's nomenclature, the first type of relationship is known as a dependency. A software package A depends on another software package B if B is required to be installed in order for A to function. An example of such dependency is the *gcc* run time library: when a program that is compiled using *gcc* is run, it requires the *gcc* run time library[7]. We will refer to these dependencies as *run-time dependencies*. The second type of relationship is known as a *build dependency*. For example, a software package might require *bison* to be built, but once the package is built (e.g. compiled) *bison* is no longer required.

3.1 Dependency Graph

We define a dependency graph as a directed graph $G = (V, E)$ where V is the set of packages, and there exists a node $(V_p, V_d) \in E$ iff V_p directly depends on V_d. Each edge is labeled according to the type of edge it is (a run-time dependency or a build dependency). G is a directed acyclic graph.

We define the set of *one level dependencies* of package p, denoted as $Dy(p)$ is a set of nodes in G such as:

$$\forall d \in Dy(p) \quad \exists \quad \langle p, d \rangle \in G$$

We define the set of *all dependencies* of package p, denoted as $Dy^+(p)$, as a set of nodes in G such as:

$$\forall d \in Dy^+(p) \quad \exists \quad \text{a path from p to d in } G$$

[7]This is independent on whether dynamic or static linking is used; in the former the library is installed independent of the software package, while in the latter the library is embedded in the software package itself.

$Dy^+(p)$ corresponds to all the packages that are required by p.

The set of *one level dependents* of package p, denoted as $Dt(p)$ is a set of nodes in G such as:

$$\forall d \in Dt(p) \quad \exists \quad \langle d, p \rangle \in G$$

The set of *all dependents* of package p, denoted as $Dt^+(p)$ is a set of nodes in G such as:

$$\forall d \in Dt^+(p) \quad \exists \quad \text{a path from d to p in } G$$

$Dt^+(p)$ corresponds to all the packages that require p.

For example, *bison* lists *gawk* as a requirement; and *gawk* lists *gettext* as one of its requirements. *gawk*, and *gettext* are elements of $Dy^+(bison)$ (*gawk* and *gettext* are dependencies of *bison*). *bison* and *gawk* are elements of $Dt^+(gettext)$ (*bison* and *gawk* are dependents of *gettext*).

We can now define the *dependency graph* of a package p, denoted as $G_y(p) = (V_y, E_y)$ as the subset of $G = (V, E)$ such that:

$$V_y = Dy^+(p)$$

$$e = (f, g) \in E_y \iff e \in E \wedge \{f, g\} \subseteq V_y$$

The *dependents graph* of a package, denoted $G_t(p)$, defined as $G_t(p) = (V_t, E_t)$ as the subset of $G = (V, E)$ such that:

$$V_t = Dt^+(p)$$

$$e = (f, g) \in E_t \iff e \in E \wedge \{f, g\} \subseteq V_t$$

The dependency graph of a package has the following property. For any two packages p_1, p_2:

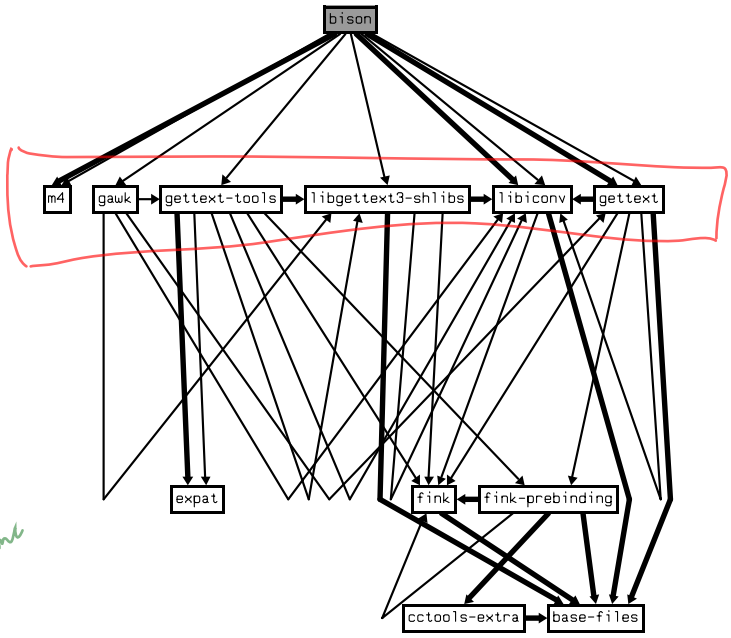
$$p_2 \in Dy^+(p_1) \implies G_y(p_2) \subset G_y(p_1)$$

In other words: if a package p_2 is a dependency of p_1, then the dependency graph of p_2 is contained in the dependency graph of p_1.

Figure 2 shows the dependency graph of *Bison*: $G_y(Bison)$. Figure 3 shows the dependency graph for *PostgreSQL*: $G_y(PostgreSQL)$. *PostgreSQL* requires *Bison* ($Bison \in Dy^+(PostgreSQL)$) and, as a consequence, the dependency graph of *Bison* is contained in the dependency graph of *PostgreSQL*.

Figure 4 shows the dependents graph of *Bison*: $G_t(Bison)$.

Most packages do not have any dependents (size of $Dt^+ = 0$), but most packages have dependencies (size of $Dy^+ > 0$). The average size of Dt^+ is 10.9 (50.9 standard deviation), and the average size of Dy^+ is 15.3 (20.9 standard deviation). The maximum size for Dy^+ is 131, and the maximum of Dt^+ is 726. There are a total of 280 packages with $Dy = 0$ (no dependencies) and 721 with $Dt = 0$ (no dependents). 162 packages have no dependencies and no dependents (for example *dict*, a "DICT protocol



run-time & build

Figure 2. Dependency graph for *Bison* $G_y(Bison)$. The node corresponding to *Bison* is highlighted. Thick edges represent run-time dependencies, and thin edges build dependencies

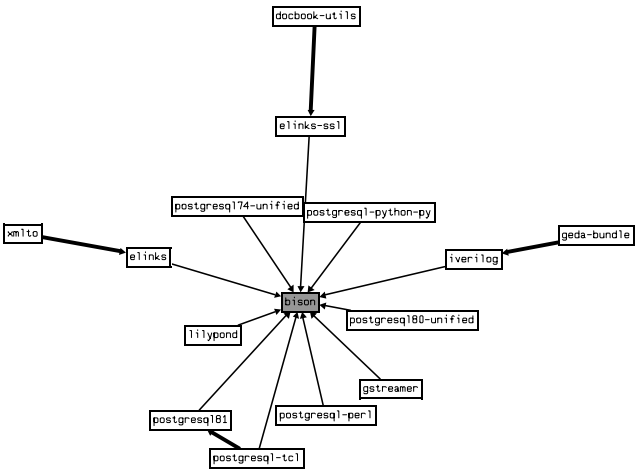
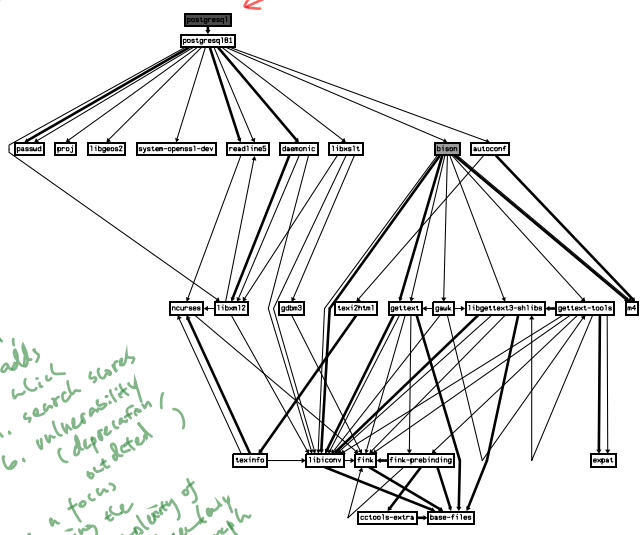


Figure 4. The Dependents graph for Bison $G_t(Bison)$

reverse dependency graph



hierarchical layout

application

based on this graph My work adds high lights ① show a. search scores b. vulnerability (deprecation / outdated)

② create a focus by reducing the complexity of dependency graph

Figure 3. Dependency graph for *PostgreSQL*. Package *Bison* is also highlighted.

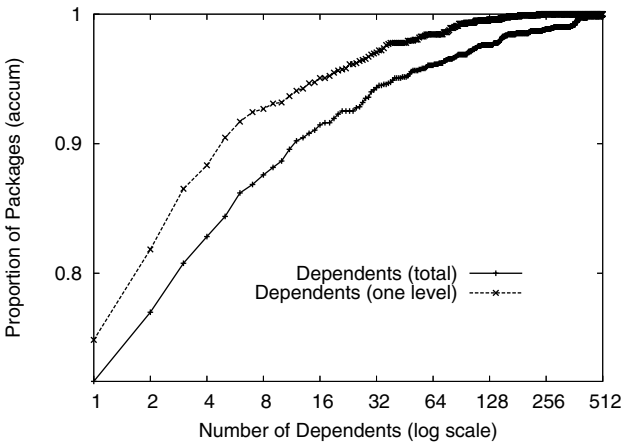


Figure 5. Accumulated distribution of the number of dependents of a package (G_t)

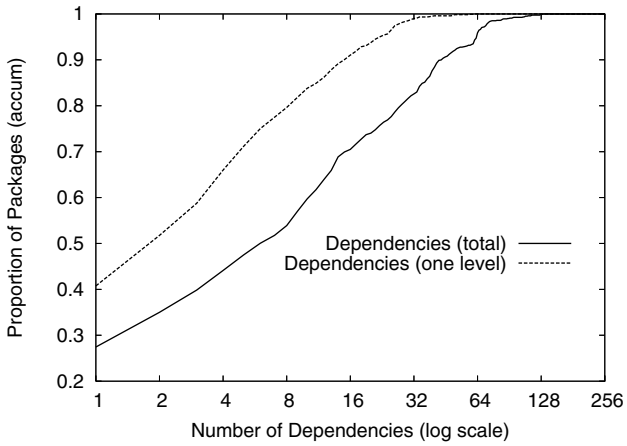


Figure 6. Accumulated distribution of the number of dependencies of a package (G_y)

dictionary-lookup client", and *dos2unix* "Convert DOS or Mac text files to Unix format".

Figure 5 shows the accumulated distribution of the number of dependents (one level and total) for all packages in Fink. For example, it shows that around 85% of all packages have 8 or less total dependents, but around 92% of all packages have 8 or less one-level dependents.

Figure 6 shows the accumulated distribution of the number of dependencies (one level and total) for all packages in Fink. It shows that around 50 percent of packages have 8 or less total dependencies, but 50 percent of packages have 2 or less one-level dependencies.

3.2 Discussion

Not surprisingly most packages have very few dependents (almost 60% have zero). This is most likely because the package is an end-user application. Some packages have very large dependents graphs, and these tend to be 1) low-level, generic domain libraries that are widely used; and 2) tools needed to build packages (for example, the packages *fink*, *cc-tools-extra*).

In terms of dependencies, it is not surprising that more than 75% of packages of Fink require at least another package.

An interesting issue is why 180 packages can exist without any dependency. This is probably because Fink requires (even before it is installed) the installation of *Xcode*[8]. *Xcode* is a development environment for OS X that provides the basic software development tools (such as compilers, make, Java developer environment, etc).

[8] http://www.apple.com/macosx/features/xcode/

4 Success

Many open source applications are undeniably considered a success. For example, according to the Netcraft survey by Dec 2006 the Apache Web server has captured 60% of the Web server market and a total of 63,819,607 Web servers use it[9].

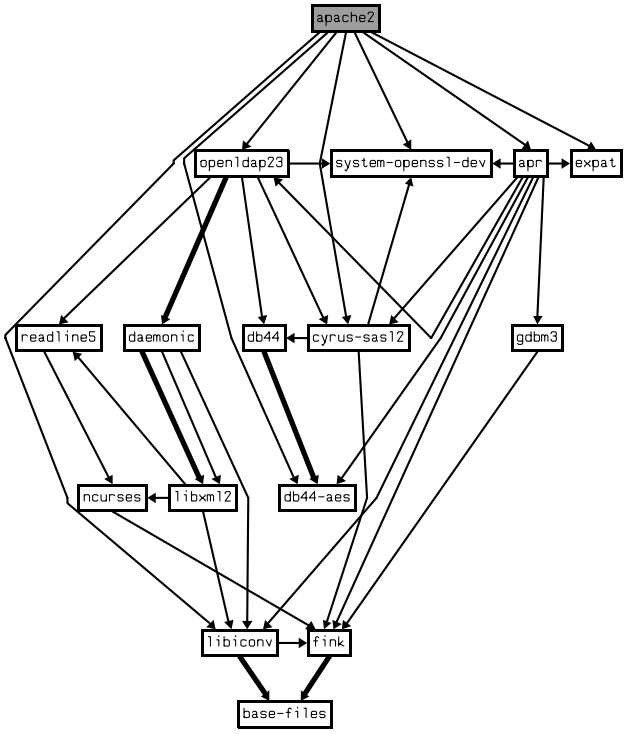


Figure 7. Dependency graph for *Apache2* $G_y(Apache2)$. The node corresponding to *Apache2* is highlighted.

The Fink package *Apache2* (corresponding to version 2 of the Apache server) requires several other applications, both to be built and run (its direct requirements are *libiconv*, *db44-aes*, *expat*, *apr*, *openssl* and *openldap*, and 15 applications in total compose its dependency graph, which is depicted in figure 7).

4.1 Transitive success

If *expat* and *openldap* are required by *Apache2*, and *Apache2* is "successful", then should *expat* and *openldap* also be considered "successful", regardless of any of its own attributes (such as market share, number of users, number of downloads, number of core developers, number of version

[9] http://news.netcraft.com/archives/web_server_survey.html

control commits, etc)[10]? In the proprietary software world success is most frequently measured in terms of revenue. If a package p licenses package d from another company some type of compensation is likely to be transfered from the owner of package p to the owner of package d. The success of p is likely to mean success for d (of course the level of success depends on the terms of the license and the contract between the two owners).

We formally define this relationship as *transitive success*: if an application A is successful, then an application R it requires (either to be built or to run) is *transitively successful* with respect to the success of application A. We denote this relationship as $\tau(R, A)$.

Notice that we have not formally defined the meaning of success. What this relationship is asserting is that, if an application is found to be successful (in whatever manner) then the applications it requires are transitively successful according to that measure of success. In the case of *Apache2* we assert that because *Apache2* is successful then *openldap* and *expat* are transitively successful with respect to *Apache2*.

One has to take into account that `.info` packages indicate one particular way in which a system is built (a decision made by the *packager*–the person who maintains the `.info` file). In the case of *Apache2 openssl* and *openldap* are optional dependencies (according to *Apache2*) but they have been chosen to be required dependencies under Fink by the packager[11].

Assume we have three applications A, R and O; R is a required dependency of A and O is an optional dependency of A. We assert that $\tau(R, A) > \tau(O, A)$ (the transitive success of R–with respect to A– is higher than the transitive success of O–with respect to A) because A can be created without O, but A cannot be created without R.

If A has two required dependencies R_1 and R_2, then $\tau(R_1, A) = \tau(R_2, A)$.

If A optionally depends on two applications O_1 and O_2, then we cannot assert that $\tau(O_1, A) = \tau(O_2, A)$. This is because O_1 might be a more popular option than O_2 (i.e. more people build or run A with O_1 than with O_2).

Sometimes a dependency is meaningful only under a particular environment. For instance, to build or install *Apache2* using Fink it is necessary to first install the package *fink* (this package contains all the necessary infrastructure to download, install, and keep track of packages). The dependency graph of *Apache2* transitively depends on the package *fink*). But *fink* is not needed to install *Apache2* outside the Fink environment (such as when using other packaging systems such as Debian, or Red Hat). This type of relationship can be interpreted as: "under the *Fink* environment the package *Fink* is transitively successful with respect to *Apache2*".

In few cases a dependency can be satisfied by one of several packages. For example, under Fink the packages *xorg* and *xfree86* are equivalent (they are both X11 servers and libraries). Typically a package contains the dependency "*xorg* or *xfree86*". From the point of view of success, both *xorg* and *xfree86* compete for the same "market" (to be the package that solves the dependency "an X11 server" for a given application). Which one is more transitively successful with respect to a given application A will depend on the proportion of users that choose *xorg* over *xfree86* when they build or run A.

Another interesting issue is when the features of a package d are not used by the package p that depends on it. For example, d provides some routines that are never used by p. It might be possible that when p is executed it never uses any feature of d. One-level dependencies might be more important than all dependencies to determine transitive success.

In conclusion, transitive success can be used to measure the success of an application based upon the success of the applications that require it. But transitive success is not a measure of success by itself and it should be used with care.

4.2 Dependency success

The dependency graph of Fink can also be used as a proxy to measure "success". According to the metrics used by SourceForge, the project *fink* is arguably successful (*fink* ranks 86 in all-time activity in SourceForge with an average of 1000 daily downloads during Dec. 2006). The only reason to download and install *fink* is because one is interested in installing other applications.

Unfortunately it is not trivial to determine what applications are installed using Fink. When first installed, Fink installs a very small number of required applications (packages); most applications are installed only when the user requests them, either explicitly –user wants the application installed– or implicitly –an application is required to resolve a dependency[12].

We can assert, however, that packages with large dependents graphs (measured by the number of edges in them) are more likely to be installed by users. In other words: the more packages that require a package A, the more likely A will be installed.

[10] *expat* ranks 2042 in the "all-time" rankings of SourceForge, while *openldap* is not even hosted by SourceForge.

[11] It is not uncommon in the open source software world to have optional dependencies; which of these optional dependencies are actually used are specified during build time; in the case of Apache this is done during the configure step with an options such as `--with-ldap=yes`. This information is included in the `.info` files in sections *ConfigureScript* and *ConfigureParameters*.

[12] The author's two computers have 465 and 333 packages installed via Fink

We formally define dependency-success of A, $s_d(A)$ as the proportion of packages (according to Fink) that require (are dependents of) package A. Given $G_t(A) = (V_t, E_t)$ and the dependency graph of Fink $G = (V, E)$:

$$s_d(A) = \frac{|V_t|}{|V|}$$

Let us order all packages in Fink using their dependency-success, in reverse order (the ones with the highest will be first). We define the dependency-success ranking of package A as the position of A in this list. Table 3 highlights the dependency-success and dependency-ranking of the top 20 packages in Fink.

Depend. Ranking	Package	Depen-dents	Depend. Success
1	base-files	726	0.60
2	fink	722	0.59
3	libiconv	426	0.35
4	libncurses5	387	0.32
5	glib	380	0.31
6	expat	379	0.31
7	ncurses	376	0.31
8	libgettext3-shlibs	366	0.30
9	cctools-extra	366	0.30
10	fink-prebinding	365	0.30
11	gettext-tools	355	0.29
12	pkgconfig	345	0.28
13	libpng3	282	0.23
14	libjpeg	272	0.22
15	gettext	253	0.21
16	readline5	244	0.20
17	libxml2	211	0.17
18	libtiff	207	0.17
20	opensp4	160	0.13

Table 3. 20 packages with the highest dependency-success.

Dependency success is strongly biased to favor libraries and tool for software development. Many applications that are widely used might not be required by another package (23% of packages in Fink have $s_d = 0$).

There are very few libraries used as subjects in empirical studies of open source software. This is perhaps because they are essentially invisible to the final user: a user does not necessarily know all the different libraries required to run a given application. Without many of these libraries the application might have never been created.

Dependency-success is a metric that should be used carefully. It only measures the proportion of applications that use a package, and it is therefore biased towards packages in generic domains.

Dependency-success can only be used reliably to compare similar packages. Assume there exist two packages that have the same goal L_1 and L_2, and that they have dependency-success i and j. If $i >> j$ we would be able to assert that L_1 is more successful (within Fink) than L_2. For example, *aspell* and *ispell* are both packages to verify spelling. *aspell* is newer, more powerful and endorsed by GNU. One of the design goals of *aspell* is to be the replacement of *ispell*. *aspell* ranking is 86 (27 packages depend on it), while *ispell* is 156 (8 packages depend on it).

The history and evolution of the dependency-success of packages can also be useful. This information can highlight if one package is being replaced by another (we plan to perform this analysis in the future).

5 License

Does the license affect the size of the dependency graph? Licenses create islands that determine whether software can be reused or not. For example, a project d can be a dependency of another p iff the license of d allows its use under the license of p. For instance, software released under BSD license can be used by software under the GPL, but not the other way around[5].

Table 4 lists of the licenses (as listed in the field License) and the proportion of packages the use them. In a preliminary analysis we did not find a significant difference in the proportion of licenses used by packages with no dependents. We expect to continue our work in understanding how licenses affect the development of dependency graphs.

GPL	43%
BSD	13%
OSI-Approved	12%
LGPL	6%
Restrictive/Distributable	6%
Artistic	6%
Restrictive	4%
GPL/LGPL	6%
Public Domain	3%
LGPL/GFDL	1%
GFDL	1%
GPL/GFDL	1%
Artistic/GPL	1%
GPL/LGPL/GFDL	1%

Table 4. Proportion of all packages by license

6 Conclusions and Future work

In this paper we have presented a preliminary study of the dependencies in FOSS and used this information to de-

Fourth International Workshop on Mining Software Repositories (MSR'07)
0-7695-2950-X/07 $20.00 © 2007 IEEE

fine metrics that attempt to identify applications that are widely used as by other applications (and therefore are successful).

We formalized the notion of dependency in FOSS, and defined transitive-success and dependency-success as metrics. The premise behind transitive-success is that if one application is successful, then the applications it depends upon should also be considered successful. Dependency-success, on the other hand, draws attention to those applications that are widely used by other applications.

The dependency graph points to the fact that FOSS applications are highly interrelated. Each application depends on other applications, and each applications might have other applications depending on it. As we described in [1] these relationships create a meta-community, where contributors and users from one community contribute (directly or indirectly) to the other communities. It is not uncommon for contributors of one project to subscribe to mailing lists in another project to gain awareness of where the project is and how it is evolving. Using the dependencies graph as a basis, we can conduct research to find out if and how knowledge flows from one community to another via its common contributors.

We plan to continue our study of dependencies among FOSS. First of all, we need to determine how accurate `.info` files are at describing the dependencies between packages (in this paper we assumed that `.info` packages in the stable version of Fink are accurate). Debian and Red Hat are two more sources of dependencies information (both contain more packages than Fink). We expect to do a similar study in both distributions. It will be valuable to compare their dependencies graphs, and the dependencies graphs of the same package in each of the three distributions. Another area of potential research involves the effect of licenses in the dependency graphs. Some licenses are not compatible, and therefore do not allow software licensed under their terms to be combined.

An extensive comparison of applications present in distributions and in SourceForge is required. Are there some metrics of success in SourceForge that correlate to metrics of success in distributions?

A historical analysis of the distributions and their dependencies graphs is also needed: do dependencies graphs evolve? If so, how? Are some applications losing or gaining dependencies? Are there applications which are replacing other applications?

Compared to SourceForge software distributions tend to contain software that is good enough for a large number of users (whether it is old or new). They frequently add and remove software to react to the current FOSS ecology. Software distributions should be considered a rich source of information about FOSS.

Acknowledgments

The author would like to thank the reviewers for their valuable comments. This work has been supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

[1] D. M. German. The Flow of Knowledge in Free and Open Source Communities. In *2nd. International Workshop in Supporting Knowledge Collaboration in Software Development (KCSD 2006)*, Sept. 2006.

[2] J. Lerner and J. Tirole. Some simple economics of open source. *Journal of Industrial Economics*, pages 197–234, June 2002.

[3] G. Madey, V. Freeh, and R. Tynan. *Free/Open Source Software Development*, chapter Modeling the F/OSS Community: A Quantitative Investigation, in Free/Open Source Software Development. Idea Publishing, 2004.

[4] G. Robles, J. M. Gonzalez-Barahona, M. Michlmayr, and J. J. Amor. Mining large software compilations over time: another perspective of software evolution. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 3–9, New York, NY, USA, 2006. ACM Press.

[5] L. Rosen. *Open Source Licensing: Software Freedom and Intellectual Property Law*. Prentice Hall, 2004.

[6] D. Spinellis and C. Szypersky. How is Open Source Software Affecting Software Development. *IEEE Software*, 21(1):28–33, Jan-Feb 2004.

[7] G. von Krogh, S. Spaeth, and S. Haefliger. Knowledge reuse in open source software: An exploratory study of 15 open source projects. In *HICSS '05. Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, pages 198b–198b, Jan 2005.

deprecation of packages.