A Model to Understand the Building and Running Inter-Dependencies of Software

Daniel M German
University of Victoria
dmg@uvic.ca

Jesús M. González-Barahona Gregorio Robles
Universidad Rey Juan Carlos
jgb,grex@gsyc.escet.urjc.es

Abstract

The notion of functional or modular dependency is fundamental to understand the architecture and inner workings of any software system. In this paper, we propose to extend that notion to consider dependencies at a larger scale, between software applications (usually programs or libraries themselves). These dependencies, which we call inter-dependencies are of exceptional importance in free an open source software (FOSS), where it is common to build new applications by taking advantage of a rich and complex environment of programs and libraries whose functionality is available. To explore this concept, a methodology and visualization for studying inter-dependencies of a complex software system is presented and applied to one of the largest distributions of FOSS: Debian GNU/Linux.

1. Introduction

The benefits of reusing pre-existing components when building a new software system are well known[12]. The idea of packaging a certain set of functionality into a "library", and later the concept of "software component" (which led to the COTS—components-of-the-shelf—paradigm) are good examples of this practice.

In the traditional software industry, most components (specially COTS) are provided to third parties in binary form only due to intellectual property constraints [4]. That means that, for practical purposes, they become black boxes, and little or no information about their internals is available outside. Probably because of that, most research on dependencies has concentrated on a) systems where all the source code for the application is available, and can therefore be analyzed; b) environments in which those dependencies are clearly publicized (such as network oriented services) and c) at the design and requirements level [5, 1, 6, 10, 3, 16].

Free and Open Source Software (FOSS) provides a different paradigm: software is available in source code form, and the potential user can download and build the software (and in some cases customize it). Building many FOSS programs is not an easy task, since many build-time dependencies (compilers, interpreters, libraries, or other modules that are expected at compile-time) have to be met. And once the ready-to-run version has been produced, executing it in another host (even with a very similar system and configuration) might be also problematic.

FOSS software has also become more complex, and the number of applications in the FOSS ecology keeps growing (in approximately 10 years the number of application in Debian, one of the most important distributions of FOSS software has grown from 1,500, to more than 10,000 [13]). FOSS developers commonly reuse other FOSS applications (in a method similar to the one promoted by COTS) [8].

An application's developers usually document what is needed to compile and install it (a process that has been simplified by systems such as autoconf and automake). It is not uncommon for these developers, however, to miss (or wrongly assume) some components. Only a real compilation in a controlled environment can assess the exact collection of software needed for building the software (although certain tools can help in this task). Run-time dependencies are also tricky, since it is easy miss a dependency which is only needed when certain path of execution is followed.

Downloading and compiling a complex application can be a daunting task. It might require downloading, building and installing many other applications beforehand. Finding these dependencies, and having a clear view of them and their impact, is not only important for running and building the application. Security and reliability implications are also important [4], since running any application implies potentially running any part of the full tree of dependencies it needs. Detection of common dependencies is also an interesting field, since some obscure, little known modules could be a dependency for many end-user applications, potentially becoming critical for their functionality or performance. FOSS is also a very specialized ecology of applications, and reuse is encouraged. We have observed situations in which an application has been split into two or more, be-

cause it is determined that some of its functionality might be useful to other projects [7]. Bringing a FOSS application into an organization means bringing a potentially large set of other applications along (required to build and run it). It is therefore important to understand what applications are required by another one.

As we argued in [8], understanding dependencies among applications is important for research. Most research has focused on a small number of end-user applications, such as Apache, Mozilla, Eclipse, and PostgreSQL, and rarely on libraries and other support applications. These applications require many other applications (libraries, and support programs) to exists, which could be argued should also be considered successful. Understanding the dependencies of successful FOSS can bring attention to other projects worth studying that lack direct end-users (such as libraries) and are therefore invisible to most users.

The notion of dependencies between applications is not only applicable to FOSS. It might also be exploited and used in large organizations that have many different teams creating relatively independent applications that are expected to work in concert.

The goal of this paper is to create a framework to model, extract, study and visualize the dependencies among applications. In section 2 we formalize the requirements that one software application has for other applications—its inter-dependencies—and describe their characteristics. In section 3 we define inter-dependency graphs as a method to formalize inter-dependencies. In section 4 we describe two methods to build these, and exemplify them by showing the inter-dependencies of Bugzilla. We end with conclusions and future work.

2. Dependencies between software applications

Most software applications verify (during their installation) for the proper environment where they are to be executed. They will verify that all the needed components, libraries and support programs are installed. The installation system might install any dependency, or ask the user to do it. Fewer verify—on a regular basis—that there exists a proper environment for them to continue to function (and will cease to work, or in a worst-case-scenario, perform an unwanted operation). Its software developer follows a similar process to create the necessary environment to build the application and it might include installing compilers, a configuration management systems, SDKs, etc.

Open source software applications, as its name implies are distributed in source code form. They can be downloaded and built, then installed. In general an open source application is built in the computer where it is going to be installed; this simplifies tracking and installing dependencies (many installation scripts of open source applications do not verify that dependencies are installed, since this is usually done at built time). In [11] Karels enumerates these as integration issues typical of creating a system based on open source software:

- Selecting and downloading an appropriate version of the software (and all its inter-dependencies).
- Compiling/building the software (including creating the necessary environment to compile the system–e.g. installing the necessary compiler).
- Installing, configuring, and testing the software where it is expected to run.

Usually the inter-dependencies of an application are described in their configuration management systems, using *autoconf*, *cmake*, *ant*, or *make*. A FOSS package will usually include a configuration file for one of these systems. This configuration file will try to make sure the building environment is sufficient to create the executables from the source code. Once the binary is created, it is assumed that it will run in the same system where it has been compiled, and (in general) the application no longer checks if the running environment is sufficient to execute the application.

This process should be performed (in advance) for each of the dependencies of the application (which themselves have other dependencies). For example, in order to be installed and function, *Bugzilla* requires *Perl* and a Web server (such as *Apache*) to be installed before it. *Apache* requires *XPat* (an XML parsing library for C), which in turns requires a C run-time library. In order to build and install a complex application it might be necessary to download, build, install and configure dozens of different applications.

A FOSS application (typically made available in source code form in a zip or tar file) might result in different "packages" that can be installed independently of each other. For example, a library (such as libjpeg[1], which implements I/O to JPEG files) can be divided into header files (needed to compile applications that use the library) and a dynamic library (needed to run applications that use the library). Anybody wanting to build an application that uses the library needs both, but once it is compiled only the latter is needed.

For this reason we will use the following terminology: an **application** is a bundle of artifacts that include the source code and other ancillary files needed to build and run such application. An application can be divided one or more **packages**, each capable of being installed (and potentially built) independently of the rest.[2]

[1] `http://www.ijg.org/`

[2] Our notation is influenced by the Package Management Systems of GNU/Linux distributions, which use a similar definition for a package.

2.1 Inter-dependencies

We formally define the **inter-dependencies** of a package as the set of packages that are required to build and execute the package, but are not distributed with the original application. Inter-dependencies (which from this point we will refer simply as dependencies) can be classified according to the following criteria (which is summarized in table 1).

1. **Type** There are two types of dependencies: *explicit*, which state the name of the package that will satisfy it (and potentially the version) and *abstract* which describe a "class" of packages that satisfy the dependency. In the case of an abstract dependency, it should be satisfied by exactly one of several packages (all of them are expected to provide the same functionality to the packages that requires it, even if their actual implementation and features varies significantly). For example, a system might require a "relational database" (the abstract dependency) that could be satisfied by any of the RMDBS systems in the market. There exist some de-facto abstract classes of packages (such as "ANSI C compilers", or "relational database management systems"); in other cases the application might explicitly state the abstract dependency by listing all the potential packages that would satisfy it.

2. **Importance**. dependencies can be *required* or *optional*. Some features of the package might only be available if a given optional dependency is satisfied, but if it is not, the package will still function (without the extra features). Required dependencies should always be satisfied.

3. **Stage** at which it is necessary. Some dependencies are needed only during *build-time* (such as a compiler, the configuration management system, or static libraries to be embedded into the executable); others to *test* it; while others are needed during the *installation* of the package (tools to modify the configuration files in the target system); finally, during *run-time* (such as dynamic libraries installed in the target system).

4. **Usage method**. Method by which the dependency is used. They can be roughly divided in *stand-alone programs* (e.g. a database management system, a compiler necessary to build the binaries), *middleware-based* (those that use systems such as CORBA, COM, httpd for intercommunication), *plug-ins* (those that require a core application with a plug-in architecture, such as the Apache web server, Photoshop), and *linkable libraries*. This is an important issue in FOSS engineering, where licensing might impose restrictions in the way a dependency is used.

Similar to the concept of a dependency, there exist **anti-dependencies**: the package cannot function if its anti-dependency is also installed (for example, the Linux kernel cannot coexist with the BSD kernel, and vicersa—one is an anti-dependency of the other).

Classification	Examples
Type	Explicit Abstract
Importance	Required Optional
Stage	Build Installation Run-Time Testing
Usage method	Stand-alone Middleware-based Plug-in Linkable library

Table 1. Classifications of dependencies

Packages have properties that are relevant when another application uses it as a dependency, such as those summarized in table 2.

1. **Version**. In some cases an application might require an exact version, in others a given version or its successors.

2. **Source**. What software application is the package being created from?

3. **License**. Licensing is an important issue in both commercial and FOSS software: to be able to use a package (to satisfy a dependency) one needs to acquire a license that permits such use (some use the term "license compatibility": the license of the dependency D is compatible with the license of the package P if the license of D allows P to use D via the expected usage-method [15]). For example, assume we are developing a package P to distribute to others; if we use a library L released under the General Public License (GPL) via linking (to satisfy a dependency) then P should also be released under the GPL. However, if P is never to be distributed to others then we will be allowed to use L.[3] Legal issues can make it difficult to determine if one can use a particular package to satisfy a dependency.

4. **Cost**. The cost might be monetary (such as the price of a license); footprint (how much memory and/or disk space uses when it is installed) or more abstract, such as how difficult/easy it is to build and install.

5. Its own **dependencies**. What packages does it require.

3. Inter-Dependency Graphs

Dependencies can be modeled as a directed graph, where nodes are explicit packages that are connected towards their

[3] The GPL is a copyright license. It only limits the ability of the user to make copies of the GPLed software, but does not limit how the software is used. See [15] for good discussion of the GPL, other FOSS licenses, and their compatibility.

Property	Examples
Version	1.2.3b
Source	Product to buy/download.
License	BSD, GPL, by seat, royalty free, etc
Cost	Memory, price, disk, difficulty.
Dependencies	What other pkgs does it need itself.

Table 2. Properties of packages relevant for their use to satisfy dependencies.

dependencies. The edges are typed according to their importance: optional and required. Abstract packages are also nodes, and they are connected to any of the packages that can "satisfy" it. A node can be further annotated with other attributes of the package. Similarly the edges can be annotated with extra information such as the cost of including the relationship. Inter-dependency graphs are not acyclic (sometimes two or more tightly related packages need to be installed simultaneously). We refer to this graph as the **Inter-Dependency Graph** (**IDG**) of a package.

We now proceed to define some terminology: the set of all potential dependencies of P—denoted as $D(P)$—is the set of all packages that might be required by P—the set of all explicit packages in $IDG(P)$. The set of required dependencies of P is the set of packages that will always be required by P—any explicit package that can be reached from P by following edges marked as explicit. The direct dependencies of a package P, denoted as $D_1(P)$ is defined as the list of possible dependencies that P states it will or it can use (the dependency is directly connected to P). For explicit dependencies this includes each of them (both required and optional), and for abstract ones it includes any explicit package can satisfy it.

In order to visualize an IDG of a package P we use the following notation, which is exemplified in figure 1:

- P is depicted as a circle.
- Explicit packages are depicted as rectangles.
- Abstract dependencies are depicted as diamonds.
- The edge that connects a package to a dependencies uses a continuous line if that dependency is required. It uses a dashed-line if it is optional.

When a package has a large number of dependencies, the graph becomes large and the number of edges explodes, making it very difficult to draw in an understandable manner. For this reason we propose two different ways to visualize an IDG: 1) its **direct IDG—DIDG**; and 2) its **simplified IDG—SIDG**.

The DIDG of a package P is computed by pruning its IDG bread-first, which starts from P; the traversing of the graph does not continue at explicit packages, but continues at abstract packages. The edges that link P to each of these other nodes are also part of the DIDG. The DIDG of P shows the dependencies that the developers of P explicitly know are required by P, and hides the dependencies of the dependencies. See figure 2 for an example of the DIDG of the same package P depicted in figure 1. The explicit packages in the $DIDG(P)$ are equal to $D_1(P)$.

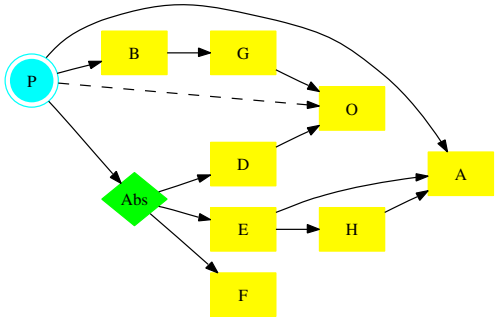

Figure 1. IDG of an application P. P has required explicit dependencies A and B, optional O (dashed lines), and abstract Abs. Abs can be satisfied by one of D, E, or F.

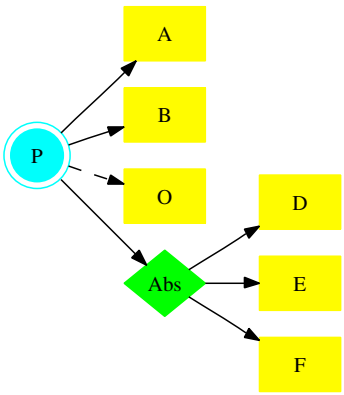

Figure 2. DIDG of an application P (see figure 1). It shows those dependencies that are directly needed by P.

The SIDG of a package P is a minimum spanning tree of the IDG of P, such that, starting from P, traverses the graph breath-first. The SIDG's most important feature is that it depicts all the potential packages needed by P, connected to the closest package that lists it as a dependency. The SIDG of our example package is depicted in figure 3. Given that there is no order in the children of a node, several SIDGs can be computed for a given IDG.

There is a variant of the SIDG that is worth mentioning.

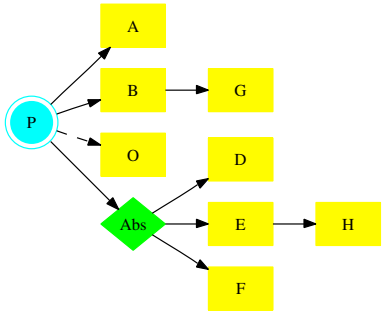

Figure 3. SIDG of P. The number of edges is reduced, but all potential dependencies are depicted, connected to the closest package to P.

In this case the minimum spanning tree is computed by first following required dependencies of explicit packages, followed by abstract dependencies, and finally, optional ones (we call this *explicit-first SIDG*) graph that can be used to determine which optional dependencies are to be required anyway (because they are required by another required dependency). For our example package P, its explicit-first SIDG is depicted in figure 4: note how O is to be required by G; this could impact the decision to include O as an option of P and in using D to satisfy Abs. It is, however, important to reiterate that SIDGs are simplifications of the IDG and do not show all the relationships between packages. This also highlights an important advantage of IDGs: their analysis can be helpful not only for reverse engineering, but for forward engineering.

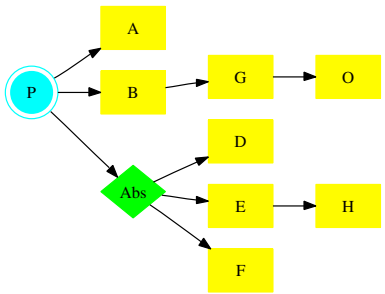

Figure 4. Explicit-first SIDG of P. Compare this graph to its SIDG (figure 2). Here O is shown as a required dependency of G while in its SIDG it is shown as an optional dependency of P.

3.1. Instance of an IDG

As we described earlier, depending on its abstract and optional inter-dependencies, a package can be built and installed in potentially different ways. The person building and/or installing the package decides which optional dependencies to include, and what explicit package to use to satisfy an abstract dependency. The instance of an IDG models such decisions.

Formally an **instance** of the IDG a package P—is defined any of the subsets of the $IDG(P)$ such that: 1) zero or more edges linking a package to its optional inter-dependency are removed (those optional inter-dependencies are not used); 2) for each abstract package, select one and only one edge starting from it, and remove all others (the abstract inter-dependency is resolved); and 3) after all edges have been removed, remove any node and edge non reachable from P (such packages are no longer needed).

The number of potential instances of the IDG depends on the number of optional and abstract inter-dependencies that exist in the IDG. From a practical point of view an instance of an IDG represents either the way a package is currently installed on a system, or a particular way to build it and/or install it based upon some criteria (for example, the "recommended" or the "default" ways to build/install the package).

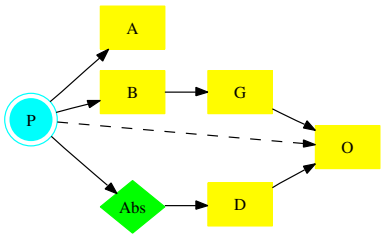

Figure 5. Instance of the IDG of P. The optional O is used by P and the abstract inter-dependency Abs has been resolved using D.

3.2. Annotating the IDG

The nodes of the IDG can be annotated according to the properties of the packages (such as version required, license, cost, etc. –as described in table 2). An instance of the graph can be build based upon a criteria that takes into account these annotations. For example, instances can be calculated that have certain properties, such as: the smallest accumulated disk space (the sum of the disk space of each of its packages), the one that does not use GPLed software (if it exists), etc.

Similarly, the edges can also be annotated based upon some of the classifications of inter-dependencies (described

in table 1). For example, an instance can be created that uses only edges labeled *Build* (representing the inter-dependencies needed to build the package, but not to run it), or that do not use CORBA (if such exists).

Being able to compute such instances could be very useful to people trying to re-engineer a system to make it run in an embedded system, where issues of footprint, and potentially licensing are important.

An enhancement to the way an IDG (or its instance) is visualized is to use the "source" annotation of the packages (as we previously described a given software application can result is one or more installable packages) create subgraphs for each source, such that they "cluster" their provided packages. This is exemplified in figure 6.

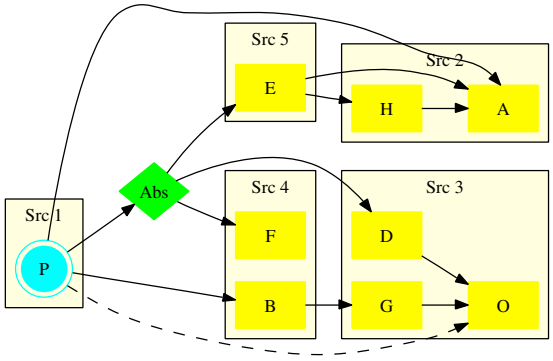

Figure 6. An IDG where an annotation of the nodes ("source") is used to show the software application where packages originate.

3.3. IDGs of a set of packages

There are occasions in which it is important to analyze a system composed of more than one package. For example, given the list of packages installed in a computer one would be interested to know their interrelationships. The IDG of a set of packages can be easily extended as the union of the IDGs of each package. The resulting IDG might contain more than one node without any incoming edges.

4. Building dependency Graphs

The most straightforward way to create the IDG of any package P is to build its IDG recursively, starting from P:

```
To compute the IDG of P:
begin
   result <- DIDG(P)
   Foreach package k in DIDG(P) do
      result <- result union DIDG(k)
   endfor
   IDG(P) <- result
end
```

The DIDG of a package can be created by inspecting the source application of the package. In some cases its developer (whether an organization or an individual) will enumerate its requirements (such as in a README, or INSTALL file). In other cases it is necessary to inspect its configuration management system files (such as those for autoconf/automake, cmake, ant, etc). Sometimes a dependency is difficult to discover because it is usually present in the environment where the package is usually built or run and it is not made explicit anywhere (particularly those that require support programs that are executed via pipes).

For example, the source code of *Bugzilla* (version 3.0) contains two installable packages: its documentation, and its actual software. It states in its file `QUICKSTART` that it requires: *perl*, *MySQL* or *Postgresql*, a mail transport agent "compatible with sendmail", and a web server that supports CGI (it recommends *apache 1.3.x*, or *apache 2.x*). It also includes a perl script *./checksetup.pl* that should be run too verify if "everything required is installed" (*Bugzilla* is primarily implemented in perl). This script (created specifically for *Bugzilla*) includes a list of optional and required dependencies (perl modules). *Bugzilla* requires three abstract packages (*DBMS*, *MTA*, and *web server*), 8 required, and 20 optional packages; figure 7 shows its DIDG.

Building the entire IDG of Bugzilla will require inspecting a very large number of individual packages. In general building the DIDG of a package is a time consuming task, and depending on the complexity of the software, a potential large number of DIDGs might need to be created.

4.1. Building DIDGs from FOSS distributions

Linux software distributions recognized this as a major challenge for the deployment and maintenance of Linux based-systems [8]. The solution was the creation of Package Management Systems (PMS) that automatize the task of downloading, building, installing, configuring, and uninstalling software applications with little or no intervention by the user [2]. Several PMSs for FOSS systems are currently in use, such as Debian's *dpkg*[4], Red Hat's *rpm*[5], Yellowdog's *yum*[6], and Fink's *fink*[7].

[4]Debian is one of the most successful GNU/Linux distributions (`debian.org`).

[5]Red Hat distributes "Red Hat Enterprise", one of the most successful commercial distributions of GNU/Linux, and Fedora Core, a non-commercial one (`redhat.com`).

[6]Yellowdog is a GNU/Linux distribution for the PowerPC architecture (`yellowdog.com`). Yum is also used by Fedora Core.

[7]Fink is a FOSS distribution for OS X (`fink.sourceforge.net`).

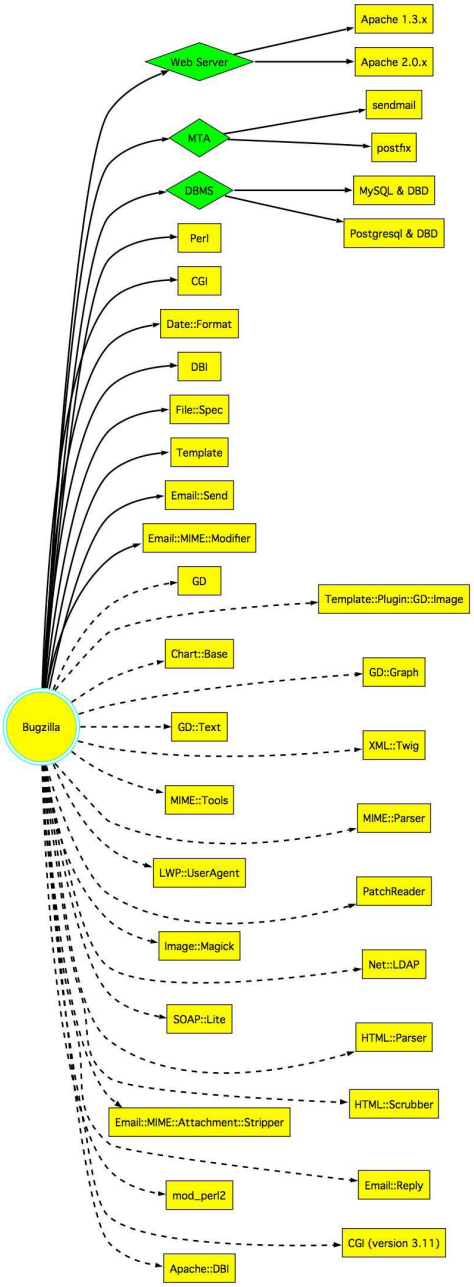

Figure 7. DIDG of Bugzilla, built by inspecting its documentation and configuration files.

The PMS requires to know (at the very least) for each package: a) the location where the software is to be downloaded (its source); b) a list of other applications that it requires before it can be built (its build dependencies), and before it can be installed (its run-timex dependencies), and the steps required to build it, install it, and un-install it. We will refer to this information as the package description.

The creation of the package description is the responsibility of the package maintainer. She is also responsible for determining the default configuration for the package, which might involve determining which optional features are to be used (or not). Her job is also to determine the build and install dependencies of the package. Some FOSS distributions use volunteers for this job (such as Debian and Fink) and others pay employees to become package managers (Red Hat).

4.2. Building IDGs of FOSS packages from Debian

Debian [14, 13] defines two different types of packages: source and binary. A source package corresponds to a down-loadable application from which one or more binary packages can be created[8]. The descriptions of each binary and source package can be found in the files *Packages.bz2* and *Sources.bz2* respectively. Figures 8 and 9 show excerpts from *Bugzilla*'s binary and source package descriptions. The fields are described in the *Debian Package Management* chapter of the Debian Reference[9].

```
Package: bugzilla
Priority: optional
Section: web
Installed-Size: 4420
[...]
Architecture: all
Version: 2.22.1-2
Depends: debconf (>= 0.9.95) | debconf-2.0,
  libtemplate-perl (>=2.10), libappconfig-perl,
  libdbd-mysql-perl, libtimedate-perl,
  libmailtools-perl (>= 1.67), libmime-perl,
  apache | apache2 | apache-perl | apache-ssl | httpd,
  sendmail | postfix | exim4 | mail-transport-agent,
  ucf (>= 0.08), patch, dbconfig-common (>= 1.8.27),
  mysql-client
Recommends: libchart-perl (>= 0.99c.pre3-0.1),
  libxml-parser-perl, mysql-server, perlmagick
Suggests: bugzilla-doc, libnet-ldap-perl,
  libgd-text-perl, libgd-graph-perl,
  libgd-gd2-perl | libgd-noxpm-perl, python, ruby
[...]
Size: 821862
[...]
```

Figure 8. Excerpt of Bugzilla binary package description in Debian 4.0

Building IDGs of all binary packages from Debian package descriptions is not trivial. It requires processing its list of binary packages (*Packages*) and its list of source applications *Sources*. Processing *Packages* and *Sources* requires two passes. In the first we create the set of all packages (explicit and abstract).

[8]The term "binary" can be misleading, as it might not contain any binaries; for example, a binary package can be composed of only documentation.

[9]http://www.debian.org/doc/manuals/reference/ch-package.en.html

```
Package: bugzilla
Binary: bugzilla, bugzilla-doc
Version: 2.22.1-2
Priority: optional
Section: web
[...]
Build-Depends: po-debconf, debhelper (>= 5), dpatch
Build-Depends-Indep: debconf (>= 0.9.95) | debconf-2.0
Architecture: all
[...]
Files:
 c5b0baf3[...]cd1df 1938535 bugzilla_2.22.1.orig.tar.gz
[...]
```

Figure 9. Excerpt of the source package description for Bugzilla in Debian 4.0

- Every binary package becomes an explicit package (from the *Packages*).
- Any name listed in a *Provides* field of package P becomes an abstract dependency (the case in which the "provided" name is the same as an explicit package is disambiguated). An edge is created from the abstract dependency to P (P satisfies the abstract dependency).
- Every time a set of options (two or more packages using | between them such as "packageA|packageB") appears in a dependency field (such as *Depends*, *Pre-Depends*, *Build-Requires*, *Recommends*, *Suggests*): create an abstract dependency (giving it a unique name), and replace such set of packages with it.

In the second pass, for every binary package description (the current package):

- For every package D listed in a *Depends* and *Predepends* field: make D a (run-time) dependency of the current package.
- For every package O listed in a *Recommends* and *Suggests* field: make O an optional (run-time) dependency of the current package.

Finally, scan *Sources*: for every source package description (the current package):

- For each package B listed in the *Binary* field:
  - Set the source of B as the current package.
  - For every package D listed in the *Build-Depends* and *Build-Depends-Indep* fields: make D a (build-time) required dependency of B.

Processing Debian 4.0 results in 18,042 explicit packages, 2,789 abstract packages are created (including the 1982 from *Provides*), and there are 10,223 source packages (which provide from 1 to 316 packages each). To generate the IDG of any given binary package P extract from the Debian graph the subgraph that is reachable from P.

The IDG of a package might not be identical from the IDG that could be extracted from the original software application. There are several reasons for this:

- Debian requires certain tools to perform the installation and setup of packages (such as *debconf*). These packages are not required if the user performs the installation from the original source code.
- The package maintainer might decide that an optional dependency is used by most of the users of the distribution, hence changes such dependency to required.
- Some optional dependencies might not be listed at all.
- Some software applications are broken into packages without the original software applications developers being aware of it. In other words: the package maintainer decides what packages are created from a given software application.
- Sometimes the source code of the application is modified (patched–in Debian terms). This might have repercussions in its IDG.
- Many packages list their requirements but do not classify them in build-time or run-time. The IDGs built from distributions usually do.
- Some intellectual property issues have affected the name of the packages in Debian (for example "Iceweasel" is the name given to the package that corresponds to a patched version of Firefox due to issues regarding trademark and copyright[10]).

4.2.1 Debian Popularity Contest

The Debian Popularity Contest is an attempt to map the usage of Debian packages. Its main goal is to know what software packages are actually installed and used. This information is used in order to determine the order in which packages are put on different CDs (i.e. packages with a high popularity and use are put on the first or first few CDs); it is also used during quality assurance activities as a criteria on which packages to focus.

We use the popularity contest data of Debian to estimate the most likely way an abstract dependency is resolved. We make the following assumption: if an abstract dependency has n-options, the one with the highest popularity is the most likely instantiation of such abstract dependency. The instance of the IDG computed using this method results in the **most popular instance** of an IDG of a package (no optional packages are included in it).

As a comparison, we present the graphs corresponding to Bugzilla according to Debian 4. Figure 10 shows its DIDG, figure 11 shows its SIDG (due to space limitations is shown in a very small size ad it only contains required dependencies). Its entire IDG has too many edges to be properly presented in a meaningful way. Figure 11 shows the SIDG

[10]See For more information visit `http://www.linux-watch.com/news/NS3364701970.html`.

of its most popular instance. In these graphs orange nodes correspond to those that are always present in a Debian system. Two interesting differences between the DIDGs of Bugzilla presented in figures 7 and 10 are that the abstract dependencies "web server" and "mail transport agent" have many more packages that can satisfy them. "Web server" can be satisfied by 22 different packages, such as *cherokee*, *boa*, *yaws*, *mini-httpd*, etc); also, the Debian package maintainer converts the dependency "*MySQL* or *Postgresql*" into *MySQL-client*, while *Postgresql* is not an option.

5. Conclusions and Future Work

In this paper we have described a method to model the inter-dependencies that are required to build, run, test, and install a given application: its inter-dependency graph (IDG). We also present a method to visualize represent them. We exemplify them by presenting the inter-dependencies of Bugzilla. We also present two methods to build the IDG of an application: one that inspects its source artifacts; and one that use a FOSS distribution (Debian).

The IDGs of a package can be used to determine variants in which the package can be installed. Their analysis can be useful (including model checking methods such as those used to model and verify components): it can show inconsistencies (such the invalid use of a packaged based upon its license), or to determine its minimum configuration (in one is interested to run inside an embedded system). From a re-engineering point of view it might facilitate the understanding of the different packages present on a system, and how they are used. When one is interested in the re-engineering of a given package the IDGs of a system can be useful to determine what packages use another one. IDGs can also be used to evaluate and audit a system. They also provide a way to identify products that compete for the same market (those that satisfy the same abstract dependency). IDGs of a distribution (and packages) can provide interesting insights into the open source ecology, where applications rarely work in isolation.

References

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.

[2] D. Blackman. Debian package management, part 1: A user's guide. *Linux J.*, 2000(80es):12, 2000.

[3] S. Dehousse, S. Faulkner, H. Mouratidis, P. Giorgini, and M. Kolp. Reasoning about willingness in networks of agents. In *SELMAS '06: Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems*, pages 91–98, New York, NY, USA, 2006. ACM Press.

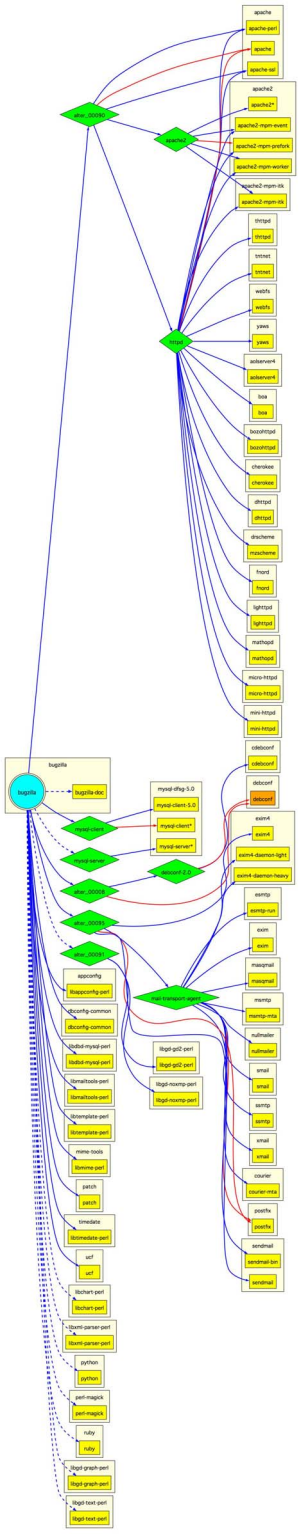

Figure 10. DIDG of Bugzilla according to Debian 4. Orange nodes are always installed, and orange edges the most popular option to satisfy an abstract dependency.

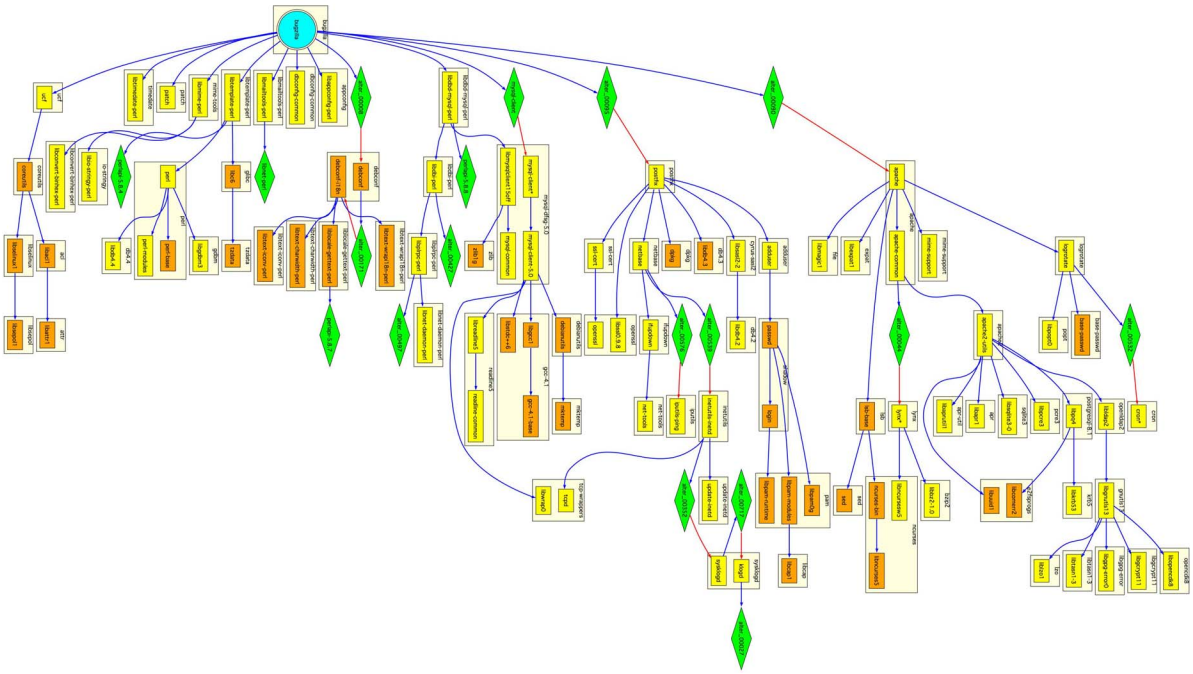

Figure 12. SIDG of Most Popular Instance of Bugzilla in Debian 4 (optional depend. not shown).

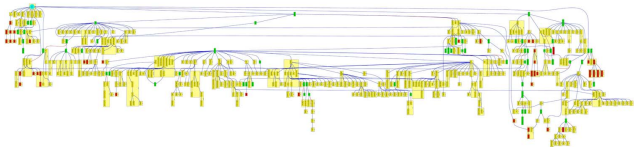

Figure 11. SIDG of Bugzilla according to Debian 4 (optional dependencies not shown).

[4] P. T. Devanbu and S. Stubblebine. Software engineering for security: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 227–239, New York, NY, USA, 2000. ACM Press.

[5] X. Franch, G. Grau, and C. Quer. A framework for the definition of metrics for actor-dependency models. In *RE '04: Proceedings of the Requirements Engineering Conference, 12th IEEE International (RE'04)*, pages 348–349, Washington, DC, USA, 2004. IEEE Computer Society.

[6] X. Franch and N. A. Maiden. Modelling component dependencies to inform their selection. In *ICCBSS '03: Proceedings of the Second International Conference on COTS-Based Software Systems*, pages 81–91, London, UK, 2003. Springer-Verlag.

[7] D. M. German. Using software trails to reconstruct the evolution of software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6):367–384, 2004.

[8] D. M. German. Using software distributions to understand the relationship among free and open source software projects. In *4th International Workshop on Mining Software Repositories (MSR 2007)*, May 2007.

[9] J. M. González-Barahona, G. Robles, M. Ortuño, L. Rodero, J. Centeno, V. Matellan, E. Castro, and P. de-las Heras. Analyzing the anatomy of GNU/Linux distributions: methodology and case studies (Red Hat and Debian). In S. Koch, editor, *Free/Open Source Software Development*, pages 27–58. Idea Group Publishing, Hershey, Pennsylvania, USA, 2004.

[10] J. Hatcliff, X. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath. Cadena: an integrated development, analysis, and verification environment for component-based systems. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 160–173, Washington, DC, USA, 2003. IEEE Computer Society.

[11] M. J. Karels. Commercializing open source software. *Queue*, 1(5):46–55, 2003.

[12] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15:1053–1058, 1972.

[13] G. Robles, J. M. Gonzalez-Barahona, M. Michlmayr, and J. J. Amor. Mining large software compilations over time: Another perspective of software evolution. In *Proceedings of the Third International Workshop on Mining Software Repositories*, pages 3–9, Shanghai, China, May 2006.

[14] G. Robles, J. M. González-Barahona, and M. Michlmayr. Evolution of volunteer participation in libre software projects: evidence from Debian. In *Proceedings of the 1st International Conference on Open Source Systems*, pages 100–107, Genoa, Italy, July 2005.

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

[16] M. Weiss, B. Esfandiari, and Y. Luo. Towards a classification of web service feature interactions. *Comput. Networks*, 51(2):359–381, 2007.