Gentoo Package Dependencies over Time

Remco Bloemen, Chintan Amrit, Stefan Kuhlmann, Gonzalo Ordóñez–Matamoros
University of Twente
PO Box 217, 7500 AE
Enschede, The Nethelands
<remco@coblue.eu> <c.amrit@utwente.nl>, <s.kuhlmann@utwente.nl>,
<h.g.ordonezmatamoros@utwente.nl>

ABSTRACT

Open source distributions such as Gentoo need to accurately track dependency relations between software packages in order to install working systems. To do this, Gentoo has a carefully authored database containing those relations. In this paper, we extract the Gentoo package dependency graph and its changes over time. The final dependency graph spans 15 thousand open source projects and 80 thousand dependency relations. Furthermore, the development of this graph is tracked over time from the beginning of the Gentoo project in 2000 to the first quarter of 2012, with monthly resolution. The resulting dataset provides many opportunities for research. In this paper we explore cluster analysis to reveals meaningful relations between packages and in a separate paper we analyze changes in the dependencies over time to get insights in the innovation dynamics of open source software.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management

General Terms

Measurement

Keywords

Innovation, dependencies, graph, Gentoo

1. INTRODUCTION

No software project stands entirely on its own. Software is usually developed by taking one or more existing libraries of components and combining those components in ways to create new products. Take for example a simple chat application. The chat application uses a different library for user interface development that provides components such as a window a text entry field and a button (that is labeled "send message" by the chat application). This user interface library in its turn uses a graphics library to draw the lines,

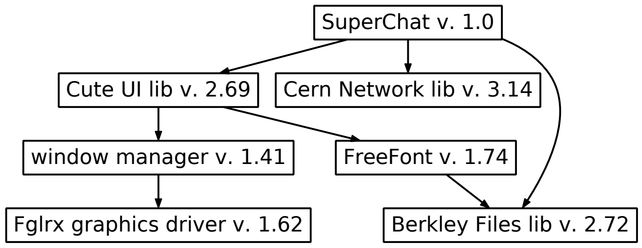

Figure 1: Example dependency graph.

rectangles and text necessary for the fields and buttons. The graphics library uses a library to read font files and uses the fonts to turn text into pictures that can be displayed on the screen. The graphics library then sends the contents of the window to the window manager, which in turns uses a graphics card driver to instruct the hardware. The chat application uses a networking library to provide it with the basic components for internet communication and uses a file library to store the users settings. The same file library is also used by the font library to read font files. The dependency graph so described is drawn in figure 1. Compared to a real chat application the graph is hugely simplified, tracing a real chat application back to all the components involved will likely result in hundreds of libraries used.

In the remainder of this article, the terms 'project', 'package' and 'library' will be used as synonyms for a node in the dependency graph.

Now that a data source is selected, it is time to extract the required information and process the data into a form that allows easy calculations. The major steps are collecting the raw data, parsing this into a simpler format and producing the final dependency graph from this simpler form. Some post-processing can then be done on the dependency graph. In the process, the dataset will shrink from thirteen gigabytes taking more than a week to collect, to thirty megabytes that can be processed in four seconds.

2. COLLECTING

The Gentoo portage database consists of a large number of text files, at least one for every version of every package, contained in a large directory structure. This entire structure is kept in a CVS revision control system that has tracked all changes to the database since the start of the project around 2000.

Using the `cvs` command one can download the entire database as it was at a certain point in history. For example

MSR'14, May 31 – June 1, 2014, Hyderabad, India
ACM 978-1-4503-2863-0/14/05
http://dx.doi.org/10.1145/2597073.2597131

the following command would download the database, as it was on 1 December 2003:

```
cvs -d :pserver:anonymous@anoncvs.gentoo.org\
  /var/cvsroot co -D 12/01/2003 gentoo-x86
```

Using a small utility written for the task, this command was repeatedly invoked to download all the databases from 1 January 2000 until today, with increments of one month. The `.ebuild`, `.eclass` and `.eblit` files are stored. Other files are ignored, to save space since they contain no relevant information. This whole downloading processes took about a week and a half and the resulting database consists of three million files occupying thirteen gigabytes of space. There are also files specifying packages renames, but since these only get appended to and never deleted they where taken from the latest version of the database.

2.1 Parsing

The files are written in a text based computer language called 'ebuild' which is based on the Bash shell script language. Being a scripting language, the files can refer to other files and include complicated code to calculate dependencies on demand. This eases the task of the script developer, since he can automate many processes, but it complicates the task of extracting data. Several approaches where tried to extract the data in the industrial quantities the analysis requires.

The first approach was to use Paludis and its C++ bindings to load a repository and extract metadata. Paludis is a tool designed to process ebuild files, we then query its database interface to gather all the dependency information from all the packages. This approach takes a lot of time, it requires around a half an hour per database version, but it fails on some of the older databases because the format of the database changed over time.

The second attempt was to use a custom build metadata extraction program that also supports an older version of the database. This parser looks for text patterns resembling dependency specifications and implements only a minimal amount of the ebuild file format (basically only dependencies and the `inherit` inclusion statement). This technique is very fast, processing the entire set in 70 minutes, but fails on the newer databases that use complex techniques such as macro's in the dependency specifications.

The final method is a hybrid of the first two, using the Paludis' `instruo` command to pre-process and expand all the macros to create simplified versions that can be parsed using the custom parser. The `instruo` command was run on every version of the database downloaded. Where the command failed on a package (the older format ones) the original was kept. The total running time of this operation was around four days.

2.2 Producing the Dependency Graphs

Now the data is in 154 snapshots of the package database in a simplified text based format. This is several gigabytes and several millions of files large and needs to be processed into dependency graphs. Obviously it is inhumane to do this by hand, therefore more specialized tools were developed.

By design the database can work with complicated dependency relations, such as "package `amarok` requires package `phonon-kde`, minimum version `4.3`, but only when feature `player` is required". This is would be coded as `player? ( >=kde-base/phonon-kde-4.3 )`. An example of the run time

```
>=media-libs/taglib-1.6.1[asf,mp4]
>=media-libs/taglib-extras-1.0.1
player? (
    app-crypt/qca:2
    >=app-misc/strigi-0.5.7[dbus,qt4]
    || ( >=dev-db/mysql-5.0.76
        =virtual/mysql-5.1 )
    >=kde-base/kdelibs-4.3[opengl?,semantic-desktop?]
    sys-libs/zlib
    x11-libs/qt-script
    >=x11-libs/qtscriptgenerator-0.1.0
```

Figure 2: Fragment of the runtime dependencies of the Amarok music player.

dependencies for the Amarok music player is given in figure 2. The figure includes more complex rules such as "either package X or package Y is required", " package X is required to contain feature Y", etcetera.

These conditional rules are relevant when compiling and running software, but the conditions are not necessary when analysing component use from an innovation perspective. If Amarok has an optional dependency on a package, the developers of Amarok are actively using the innovation provided by that package even though it may not be enabled by the end user in the final product. For this reason, and of simplicity, all the conditionals are ignored when parsing the ebuilds.

When a database snapshot contains several versions of the same package, only the latest is used. For some of the analysis techniques presented in the version information might be relevant, but at this point it was decided not to include different versions to simplify processing analysis. When several versions where available in the database at a certain point in time, only the highest version was included. This ensures that each snapshot represents the state of art at that time.

In the Gentoo Portage database the dependencies are sorted in two kinds, compile time dependencies and runtime dependencies. The first kind are required to build and install the package. The second kind is only required when actually using the package. The reason for this distinction is a technical: it solves installation issues with cyclical dependencies. For the current purposes both kinds of dependencies are considered equal.

Sometimes, package get renamed or moved around in the database, this needs to be accounted for. Luckily, to allow users to upgrade from an older version to a newer version the processing of moves and renames has been automated in the Gentoo Portage database. The developers maintain a list of all the moves and renames that have happened in the database in a structured format. The latest version of this list is used to retroactively change all the package names in the historical snapshots to their modern name. This ensures that moves and renames do not harm historical continuity in the dataset.

Using all the observations and choices made above, a tool was developed to extract dependency graphs from all the simplified database snapshots. The shear scale of the resulting dataset, 1.3 million packages and 6.9 million dependency relations, required a solution to store efficiently. Therefore, a compressed format was developed that only stores the changes between the historical snapshots instead of storing whole snapshots. The extraction process took three hours

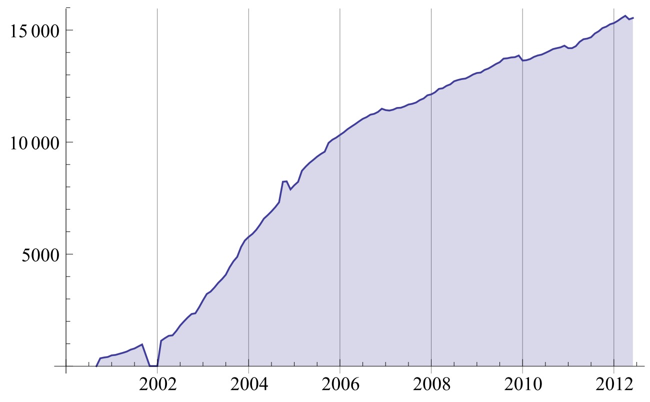

Figure 3: Growth of the total number of packages in the Gentoo package database over time.

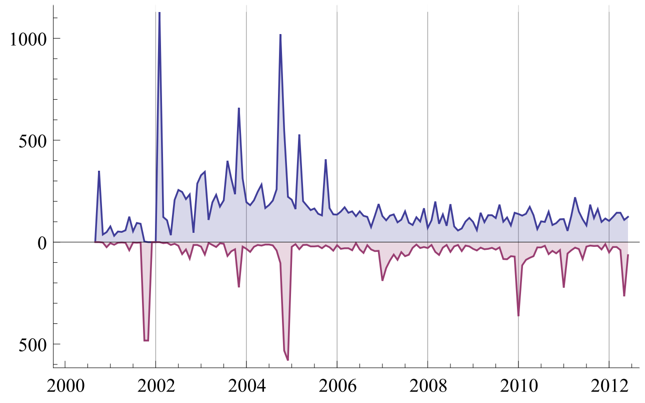

Figure 4: The number of package additions and removals over time.

and resulted in a 29 megabyte file. Reading this file into memory resident data structures and performing simple queries takes about 4 seconds. The dataset is now in a usable form.

3. EXPLORING THE DATASET

In figure 3 the number of packages in the database is plotted over time. One can see how the database started in 2001, underwent a period of rapid growth between 2002—2006 and settled into calm a linear growth from 2006 onwards. In figure 4 the rate of introducing and removing packages is plotted over time. This show two spikes, one at the start of 2002 and one in 2005. The cause of these spikes was not thoroughly investigated, but a likely cause is a massive cleanup and refactoring of the database. This is a warning sign that the data exactly at these points might contain a lot of noise. In general, the data before 2006 should be considered with more caution than the data afterwards.

The effect of the growth of the entire dataset on the number of dependencies for individual packages was investigated, but no influence was found. Since the total number of packages grows one would expect the 'market' for a certain package to grow and thus the number of dependency relations to that package to grow. To compensate for this one could divide the number of dependers by the total number of packages, compare this with using a market-share instead of an absolute number of users. In practice, this only made any significant difference for early data, but that was determined to be unreliable anyway. In the interest of keeping the analysis simple no compensation was made for the growth of the number of packages.

One of the first thing attempted after generating the dataset was to visualize the entire graph of the latest snapshot. The problem is, to make any sense of a graph it has to be laid out visually on a plane, nodes that are connected should be placed close to each other so that connecting lines are short and have little overlap. Software packages such as Graphviz, Tulip, Gephi, Jetty and Cytoscope have been tried, but after days of trying and many hours of calculation, none where able to produce any insightful layout for the sixteen thousand nodes and hundred thousand relations.

Since it was impossible to get a visual overview of the entire dependency graph, its structure was plotted using double logarithmic histograms. It was found that there there are many packages with only zero, one or a few dependencies and a few packages with a lot of dependencies. Statistically this means the distribution of the number of dependencies has a fat tail. Likewise, the number of dependers for each package was plotted, this can be interpreted as the distribution of the number of adopters of a given technology. Again, there was a fat tail, even fatter than the one from the number of dependencies. The histogram was approximately linear in the double logarithmic scale, suggesting a power-law like distribution of the number of adopters for a given technology. Zheng et al. [2] suggest that the structure of package dependency networks is not comparable to known models of social networks and have developed their own model to explain the graph structure.

4. THE KDE SUBGRAPH

The entire graph might be difficult to visualize, but a small part should be easier. The problem with choosing a small part is that the part must have a meaningful boundary, a random selection will likely have few relations and miss some key packages. It was therefore decided to only pick the packages that belong to the KDE desktop environment. The primary reason is that the author is familiar with this set of packages and knows its structure in some detail. A secondary reason is that the packages are developed by a tightly connected community where component reuse among the projects is stimulated by creating libraries. The selection was implemented by considering only packages in the category `kde-base` from the Gentoo Portage database.

The program Tulip was used to visualize the graph, the result is in figure 5. First the graph was laid out using a force based method, this clusters packages that are strongly connected close to each other. Then the graphs where colored according to their k-core measure, this is a measure for the 'connectedness' of a package. At this point there was still an unclear mess of lines between the packages, this was resolved by bundling the edges. Edge bundling merges neighboring lines to a single thicker line, this creates a vein-like structure.

In figure 5 one can see that all the packages depend on `kdelibs`, the large blue dot in the middle. The `kdelibs` package provides a lot of basic functionality, such as a unified set of icons, file open/save dialogues and less visible standard components. Almost all the packages in the KDE set require one or more of these components. It should be stressed that there was no manual work involved in the layout of this graph, Tulip was able to determine using only objective,

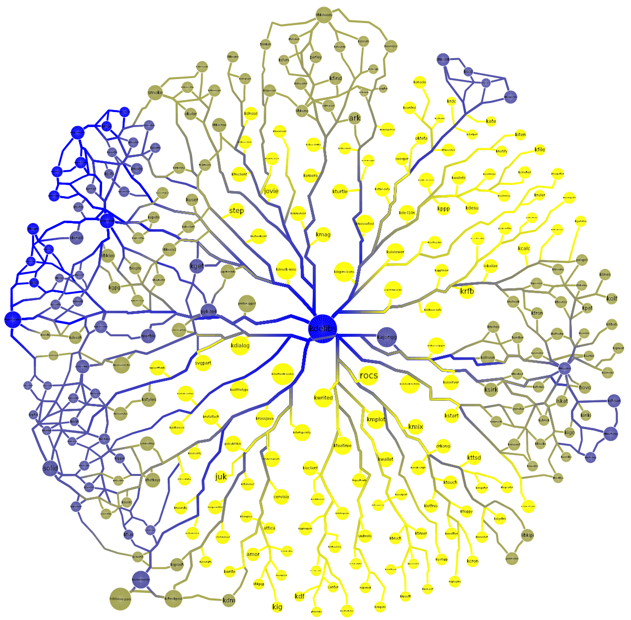

Figure 5: Internal dependencies of modules in the KDE project. Color represents the k-core measure. The graph edges have been bundled to improve readability.

deterministic mathematical methods from graph theory that `kdelibs` plays a central role in the KDE technology.

The second thing to notice are the clusters that form along the edge of the figure. All these clusters represent related areas of technology within KDE. The brownish-grey cluster immediately at the top contains mostly educational software and a few file utilities. Going clockwise, the little blue cluster next to it contains programs for compact discs. The large brownish-grey cluster on the right consists exclusively of games and supporting technologies. The complex mesh that starts around seven o'clock begins with technology used to allow users to log in. It then proceeds towards hardware related technology and desktop infrastructure. The big blue dot marked 'solid' at eight o'clock is KDE's hardware abstraction layer. At nine o'clock the big blue dot represents the notification library, used to notify users of hardware events ("battery low" and the likes), appointments or incoming emails. The mesh now shifts towards personal information management at ten o'clock. These contain utilities such as an email client, a note taking application, a chat client and a calendar application and related technologies. Lastly, the small brown-grey cluster at eleven o'clock contains technology to allow integration of scripting languages.

Scattered throughout the figure are yellow dots containing packages that are only connected to `kdelibs`, without any apparent pattern in their location. This was expected since the packages only depend on `kdelibs` and are not depended upon by other packages. This means there is no information that brings any insight in their nature and where to cluster them. Perhaps if dependencies from outside the KDE subset where included the packages would form more clusters.

It is remarkable how only a few dependency relations provide sufficient clues for the clustering algorithm to automatically find related areas of technology. Similar but faster

Table 1: Key figures of the dataset

	Time
Period	2000 to Q1 2012
Resolution	Monthly
Snapshots	154 Graphs
	All graphs
Nodes	1.3 Million
Vertices	6.9 Million
	Final graph
Nodes	15 Thousand
Vertices	80 Thousand
	Raw ebuilds
Format	SquashFS
Size	239 MB (compressed)
	4.4 GB (uncompressed)
Files	2,990,722
	Compressed graphs
Format	Tar+XZ compressed GML
Size	2 MB

clustering techniques where used on the whole snapshot with similar results. Related packages for certain programming languages (Perl, Php, Java, Python, Ruby) would cluster and packages related to either KDE or GNOME would cluster, among many more. Unfortunately the analysis software was struggling with the size of the dataset and the full set was not investigated further.

5. FINAL REMARKS

The dataset collected is available at
`http://datahub.io/dataset/gentoo-dependency-graph`
as compressed GML graphs for every snapshot. Raw data is available at
`http://www.utwente.nl/mb/iebis/staff/amrit/timeline.squashfs`
as a SquashFS compressed filesystem. The SquashFS format allows one to mount the compressed data as a read-only disk drive and operate directly on the dataset. It contains the pre-processed ebuild files and the file `graphs.bip` that contains all of the graphs in a custom format specialized for quick processing of large graphs over time. A C++ toolkit to work with the `graphs.bip` file is available at `https://github.com/Recmo/depgraph`.

In a separate paper [1] we analyzed the changes in the dependency graph over time. In particular the growth of the number of dependers on a given package is explained using the Bass model of innovation diffusion.

6. REFERENCES

[1] *Innovation Diffusion in Open Source Software.*

[2] X. Zheng, D. Zeng, H. Li, and F. Wang. Analyzing open-source software systems as complex networks. *Physica A: Statistical Mechanics and its Applications*, 387(24):6190–6200, 2008.