Large-scale code reuse in open source software

Audris Mockus
Avaya Labs Research
233 Mt Airy Rd, Basking Ridge, NJ 07901
audris@avaya.com

Abstract

We are exploring the practice of large-scale reuse involving at least a group of source code files. Our research question is to determine the extent of such reuse occurring in open source projects, to identify the code that is reused the most, and to investigate patterns of large-scale reuse. We start by identifying a sample of projects involving all code in several large repositories of open source projects, all projects bundled with popular distributions of Linux and BSD, and several large individual projects. In the next step we obtain the source code and identify groups of files reused among projects and determine the code that is most widely reused in our sample. Our findings indicate that more than 50% *of the files were used in more than one project. The most widely reused components were small and represented templates requiring major and minor modifications and a group of files reused without any change. Some widely reused components involved hundreds of files.*

1. Introduction

The practice of large-scale reuse is of tremendous interest because of the potential savings in effort and improvements in quality and lead time such reuse may bring. If the highly reused source code tends to have better quality and requires less effort to maintain [9, 3], the extent of reuse can serve as a guide of source code's reuse potential and be used to rank functionally relevant candidates [5]. Furthermore, if highly reused code and projects have attributes that distinguish them from the low reuse projects, some of these qualities may guide new projects that strive for their code to be reused more widely. Finally, existing projects may be able to take measures to increase the reuse potential of their code.

One of the advantages of open source projects is that the source code can be taken and modified by other projects in case the project is no longer supported or if the required modifications are outside the scope of the original project. It is, therefore, of interest to quantify the extent of such code sharing in practice. Such reuse may extend to projects that do not follow the open source model as well [4], though we are not attempting to quantify it here.

Our primary objective is to identify and quantify large-scale reuse in open source software. Our interest in large-scale reuse is based on interest in larger components that are more likely to contain a more complete set of functionality and, therefore, are more likely to be utilized via their interfaces as opposed to being extensively modified. In this work we investigate only one aspect of reuse where large amounts of code is copied. Discussion in Section 5 considers the limitations imposed by such approach.

To pursue our objective we first design a large sample of open source projects, retrieve their repositories, identify reused components, and quantify the extent of reuse and investigate patterns of reuse for the most widely reused components.

Section 2 considers related work, Section 3 describes methodology we have applied and Section 4 describes our findings. We conclude with discussion and future work in Section 6.

2. Related work

Despite several decades of extensive and fruitful research related to software reuse we are not aware of work attempting to quantify and investigate reuse in open source software. There are, however, numerous approaches and results closely related to our research objective. A large field of code clone detection (see, for example, [6, 7, 1]) is focused on developing better and more effective tools to detect copies of the source code. The objective of that work is focused on detection of copying of relatively small snippets of code and, therefore, is relatively expensive in terms of computational resources. Many of the tools rely on extraction of a syntax tree and, therefore, may not be easy to apply for more exotic languages and instances where the code may not parse.

Our focus is on large-scale reuse that, in principle, may

be detected by such tools, but the extent of computation needed to process large volumes of source code may be prohibitive. For example, just the source code in the CVS bundles of the 7238 projects from SourceForge represent 9.2 million versions of 2.6 million source code files containing 732 million lines of code.

It is worth noting that use of open source code in commercial projects imposes a number of responsibilities on the user and detection of such reuse represents business opportunities. Several companies, for example, Palamida (www.palamida.com) and Black Duck (www.blackducksoftware.com), have a business model based on providing tools and consulting in this area. However, their objectives are primarily focused on compliance with licensing requirements and methods are not transparent because, presumably, they represent their competitive advantages.

Google Code Search tool (codesearch.google.com) allows search of publicly available source code by language, package name, file name, and types of license. The service helps finding reusable code and, presumably, will be supported by advertising revenue as are other types of search services provided by the company. The size of the repository and the ranking algorithms are not published.

3. Method

In this section we describe our approach to achieve research objectives. We start by describing the method used to select a sample of open source projects and continue with methods to identify and measures to quantify reuse.

3.1. Sample selection and retrieval

To get a representative set of open source projects we started by selecting a number of well-known large projects including Apache, Gnome, KDE, Mozilla, OpenSolaris, Postgres, and W3C repositories that represent system, server, and user interface software. To add breadth to the list of projects we have also included all applications in several large distributions of Linux and BSD including Fedora 6, Gentoo, Slackware, FreeBSD, NetBSD, and OpenBSD. We also selected all projects utilizing version control in open development portals of Savannah, SourceForge, and Tigris. Finally, we selected projects that may not be using version control tools by obtaining the source code from repositories including FreshMeat, CPAN, RpmForge, and Gallery of Free Software Packages (www.netsw.org).

After the sample of projects was selected, it was necessary to retrieve their version control repositories or code snapshots for further processing. Unfortunately not all of the repositories make that task trivial by placing all relevant data in a single location and various tools had to be written to retrieve projects' source code.

3.2. Identifying large-scale reuse

Our definition of large-scale reuse involves actual incorporation of the parts or an entire tree of source code from a different project. It does not involve reuse at the binary level or use of the functionality from another project when the user has to retrieve the binaries or the source code for that component separately. We are looking for instances where at least several source code files are being borrowed from another project.

To accomplish the task we use the technique evaluated in [2]. The algorithm identifies reuse by finding directories of source code files that share several file names and the fraction of shared file names to the total number of filenames exceeds a specified threshold.

3.3. Quantifying reuse

Our primary question relates to the extent of large-scale reuse that we estimate by calculating the fraction of the source code that appears in more than one project. This requires operationalizations for measures counting the source code and ways to identify a project.

We chose to count the fraction of files that are shared among projects as the overall measure of reuse following empirical studies conducted at NASA [8]. Weighting such number by the size of the source code files and counting the fraction of shared components (the groups of files) are two obvious alternatives.

The extent of reuse for a particular component is simply the number of projects where its source code is incorporated.

The methodology needed to identify a project is a bit more tricky. We try to be as pragmatic as possible and define projects as the bundles in the form of a tar files or other similar packaging (e.g., rpm files in many Linux distributions, or CVS bundles representing entire version history of the project). Unfortunately, the source code bundle for the same project may be located in several places and also may have variation in the way it is named, for example, to indicate a particular released version of the project's code. Since our goal was not to count the number of places the project's code may be residing in, but rather, its incorporation into other projects, we needed a way to arrive at a canonical representation for the many possible locations and for the variations in the bundles' name.

Therefore we strip the url of the bundle's location and bundle's version number from it's name. If such transformation is not done, the bundles located at several urls

or bundles that have several version numbers would be counted as separate projects.

In particular, we would map various versions of Linux kernel repository, for example, "linux-2.4.19", "linux-2.4.33.3", "linux-2.6.18" and so forth to "linux" as a root directory of the Linux kernel project.

4. Results

As a result of our sample selection and retrieval we have obtained 13.2 million source code files. Many of these files had their version histories but versions within the history are not included in the file count. These files came from 49.9 thousand distinct bundles. In order to group multiple bundles into projects we stripped their version numbers and other variations in their names. This left us with 38.7 thousand unique projects and 10.7 million distinct file name paths. In order to focus only on text files (project repositories often also contain images and other types of binary files) we have further narrowed down the sample to 5.3 million unique file name paths representing only text files.

Filename	Frequency
Makefile.am	61161
index.html	19851
package.html	10693
__init__.py	7581
aclocal.m4	7109
index.php	6097
build.xml	5800
main.c	5552
README.txt	5335
package-frame.html	4667
readme.txt	4622
package-summary.html	4583
package-tree.html	4476
ltmain.sh	3867
main.cpp	3844
config.h	3530
package-use.html	3024
style.css	2735
test.pl	2681
acinclude.m4	2461
resource.h	2343
Makefile.inc	2304
acconfig.h	2294
autogen.sh	2277
script.xlb	2250
CMakeLists.txt	2202
main.php	2050
admin.php	1903
modinfo.php	1844
util.c	1784

Table 1. The most frequent file names.

It may be of peripheral interest to investigate the frequency of various file names. We present the most frequent file names and their frequencies in Table 1.

The 5.3 million unique file name paths were used to identify shared components as described in Section 3.2. In particular all possible pairs of directories were compared by identifying filenames that are common to the pair. The most frequent filenames that occurred more than 1000 times in the sample were excluded from the comparison. If two projects share such a file name that does not necessarily imply that projects are similar, because such frequent filenames often represent a functional role for the file (e.g., "main.c"), but do not increase probability that the file was reused. In particular, all filenames listed in Table 1 are not considered when identifying reuse because they all occur in more than 1000 projects.

If a directory pair had more than five shared filenames and if the fraction of shared filenames was greater than 80% of the total number of files in the smaller directory we considered all of shared filenames to represent code reused between the directories. The minimum of five shared filenames serves as a limit on how small the reused component may be and also reduces the likelihood that the two directories are classified as reused simply by chance. The limit on the fraction of the shared files serves primarily to exclude incorrect identification of reuse. Based on experience in [2] the particular choices tend to exclude cases where reuse is erroneously indicated, however it tends to miss groups of files that are reused from a smaller directory in a larger directory. To provide a range of the extent of reuse we have calculated the measures using three distinct cutoffs for a minimum of 80%, 50%, and 30% of the filenames shared between the pair of directories in order to consider these shared filenames to have been reused. The extent of reuse is presented in Table 2. The results do not appear to depend strongly on the algorithm parameters. Approximately half of the 5.3 million files are used in more than one project. As we discussed, we are considering only fairly large chunks of code consisting of more than five files. It is likely that reuse at finer code snippet level is even higher. This tremendous amount of large-scale reuse may represent a key advantage of open source software projects. In our experience, the reuse in commercial projects does not reach such a high level and only relatively large organization may have opportunities to reuse so much code within the organization itself.

	Reuse (30%)	Reuse (50%)	Reuse (80%)
Count	2,837,233	2,782,339	2,654,977
Fraction	.53	.52	.49

Table 2. Reused files in open source projects.

We are also interested in the most widely reused components. The most widely reused components may be in some ways distinct from less utilized ones and suggest types of source code that are most suitable for reuse.

The most widely reused set of files were language translations for user messages. The top cluster (gnome-media/po) of such translation files involved 657 projects (here and below we present cluster numbers based on the 80% cutoff only. The size of the cluster increases significantly as the cutoff is reduced). In some sense these files are not represent what is usually considered as a source code, because the translation files contain a simple template for each language. Each file contains pairs of strings used in the application — a string in English, and a corresponding string in the target language.

The next most common component with 576 instances of reuse is install module for Perl. A typical content involves the following files:

```
Module/Install.pm
Module/Install/AutoInstall.pm
Module/Install/Base.pm
Module/Install/Can.pm
Module/Install/Fetch.pm
Module/Install/Include.pm
Module/Install/Makefile.pm
Module/Install/Metadata.pm
Module/Install/Scripts.pm
Module/Install/Win32.pm
Module/Install/WriteAll.pm
```

Unlike translation files in this case the underlying functionality is very similar among the projects. Each project takes the component and modifies it to satisfy its needs.

The third most widely reused component (547 projects) involves C language functions related to internationalization. An example may be found in "a2ps" project:

```
a2ps/intl/bindtextdom.c
a2ps/intl/cat-compat.c
a2ps/intl/dcgettext.c
a2ps/intl/dgettext.c
a2ps/intl/explodename.c
a2ps/intl/finddomain.c
a2ps/intl/gettext.c
a2ps/intl/gettext.h
a2ps/intl/gettextP.h
a2ps/intl/hash-string.h
a2ps/intl/intl-compat.c
a2ps/intl/l10nflist.c
a2ps/intl/libgettext.h
a2ps/intl/linux-msg.sed
a2ps/intl/loadinfo.h
a2ps/intl/loadmsgcat.c
a2ps/intl/localealias.c
a2ps/intl/textdomain.c
a2ps/intl/xopen-msg.sed
```

Unlike the other two components, this component is reused without any modifications.

If we look for the largest components reused at least 50 times the 701 include files representing interfaces for the Linux kernel and glibc/sysdeps/generic directory containing C-language system functions with 750 files are by far the largest. Both are reused with no or little modification.

5. Validity

There are a number of threats to validity in our investigation. First question involves representativeness of the project sample. Since we do not have an infrastructure of crawling a significant portion of the web like, for example, Google Code Search, our sample may not be representative of all open source projects. Therefore, the extent of reuse in our sample may differ from the extent of reuse in all open source projects. To address this threat to validity we have designed a sampling process that incorporates source code from large and widely known projects to a much larger set of projects that are included in various operating system distribution. We also try to capture the broadest set of projects from large project repositories. Because we are primarily interested in more mature projects that use some sort of version control we may, in fact, be capturing a significant subset of the entire sample. It is, however, important to note that our results apply to more commonly used projects that are included in operating system distributions and projects that are available from the largest open source repositories.

The second major issue represents the definition of reuse. Some projects may not incorporate the source code but reuse components in binary or source code that needs to be retrieved separately (from the original project). This is clearly a desirable form of reuse because the reused code can be maintained in the centralized location. It may be that the projects that were not responsive to bug reports and/or did not parameterize well for wider use were reused by borrowing their code, while projects that were responsive and had flexible interfaces were reused without borrowing their code. Therefore, the interpretation of our results should be done with the understanding that the code is copied and possibly modified with all the potential negative (or positive) consequences that may imply.

Apart from the two main threats to validity, we assume that large-scale code reuse can be identified by detecting directories that share a large fraction of their filenames. Simple renaming of the files would make such detection impossible. Therefore, our results should be interpreted as the lower bound on the extent of reuse. The previous study on commercial code [2] found that in instances where large fraction of filenames are shared across directories the files are almost always reused.

6. Discussion and future work

Even though it is well known that a significant advantage of open source software is the ability to reuse the source code, we did not expect to observe reuse at such a large scale. It is worth noting that it is a lower boundary of reuse because we consider only instance of large-scale reuse and our measures may be missing instances where the filenames are modified in the process of reuse.

Another surprise was relative dearth of the open source software. We did focus on more mature projects that have their version control data available, and did sample projects instead of covering entire web, but even with these caveat we found only 5.3 million files in 38.7 thousand projects. In our experience commercial projects tend to be orders of magnitude larger and a comparable number of files can be observed within a single large software organization.

The three patterns of reuse we have detected do not necessarily suggest strategies projects need to undertake in order to increase reuse of their code. The first two patters represent reuse of templates (for translation and for installation), while the third represents very well defined and unchanging functionality needed to deal with international characters. Reuse of Linux kernel and gcc headers simply reflect their extensive use by other projects.

While such proliferation of code copying may appear as a bad practice, it may serve an important need in the evolutionary development of the open source software. Functionality that is incorporated in many projects can evolve and the best instances would eventually dominate (as may be the case in the third pattern of reuse we have identified). Proliferation of the same or similar code in many projects increases robustness of the entire system — the needed functionality will be more likely to survive (in terms of being maintained) in at least one project and is less likely to be affected by fates of individual projects.

Clearly, this research is exploratory and can be extended in several directions. A number of improvements in project sampling strategy can be applied to improve coverage. The precision of the currently approximate technique to detect reuse can be improved using various clone detection techniques. More importantly, better ways to define reused components and automation of techniques that detect various patterns of reuse could be designed.

The most interesting directions, nevertheless, involve extending and rephrasing research questions. In particular, we are interested in differences between reuse in closed and open source projects, and, more importantly, in the relationship of reuse and key software engineering questions related to reduction in effort and lead time and improvements in product quality. In addition to immediate cost reductions and quality improvements, the reuse may also have significant social benefits within an enterprise by connecting development groups, within open source community by serving as a basis for developer networks, and by providing links between enterprises and open source community.

7 Acknowledgments

We gratefully acknowledge support by the National Science Foundation to several collaborators of this study under Grant No. IIS-0414698.

References

[1] B. Baker. On finding duplication and near duplication in large software system. In *IEEE Working Conference on Reverse Engineering*, 1995.

[2] H.-F. Chang and A. Mockus. Constructing universal version history. In *ICSE'06 Workshop on Mining Software Repositories*, pages 76–79, Shanghai, China, May 22-23 2006.

[3] P. T. Devanbu, S. Karstu, W. L. Melo, and W. Thomas. Analytical and empirical evaluation of software reuse metrics. In *ICSE 1996*, pages 189–199, 1996.

[4] R. Ghosh. Final report. study on the economic impact of open source software on innovation and the competitiveness of the information and communication technologies (ict) sector in the eu. Technical report, UNU-MERIT, NL, 2006.

[5] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component rank: Relative significance rank for software component search. In *ICSE'03*, pages 14–24, 2003.

[6] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token- based code clone detection system for large scale source code. *IEEE Trans. Software Engineering*, 28(7), 2002.

[7] C. Kapser and M. W. Godfrey. Improved tool support for the investigation of duplication in software. In *International Conference on Software Maintenance*, 2005.

[8] F. McGarry, R. Pajerski, G. Page, S. waligora, V. Basili, and M. Zelkowitz. Softwre process improvement in the nasa software engineering laboratory. Technical Report SEI-95-TR-22, CMU, December 1994.

[9] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz. n empirical study of software reuse vs. defect-density and stability. In *ICSE 2004*, pages 282–292, 2004.