Longitudinal Analysis of Long-Timescale Open Source Repository Data

Bart Massey
Computer Science Dept.
Portland State University
Portland, Oregon USA 97207-0751
bart@cs.pdx.edu

ABSTRACT

One of the more unique features of open source software development is the continuity of projects over large time scales and incremental development efforts. For this reason, the open development process provides an interesting environment for investigation of the software development process. The problems of data collection and analysis of two particular long-running repositories, the X Window System and the Nickle Programming Language, are considered here as instructive examples. The use of uniform software tools (CVS/RCS) with open formats and interfaces makes it possible to collect data that provide unique analysis opportunities.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.8 [**Software Engineering**]: Metrics

General Terms

Measurement

Keywords

Open Source, Source Repositories

1. INTRODUCTION

When considering continuity of projects over large time scales, open source comes immediately to mind. Open source codebases tend to evolve incrementally over time instead of through catastrophic rebuilding. This is at least partly due to the lack of a need to "re-sell" software to maintain a revenue stream, resulting in greater acceptance of incremental user-visible change. In addition, while management of a codebase may change hands as the result of a "fork", the developer pool tends to change gradually—mass exoduses or arrivals of developers are rare. The likelihood of investment of current developers in the codebase decreases the inclination to discard and rebuild large chunks of code without careful consideration.

PROMISE'05 May 15, 2005 St. Louis, Missouri, USA

This (more) gradual process is coincident with the widespread use of well-understood repository software, collaborative development models, and of course the openness of both code changes and development process. As a result, the longitudinal study of long-standing open source codebases by automated repository data extraction promises to give insights into code evolution and maintenance that would be more difficult to achieve using commercial data, even if it were available.

The datasets that are being developed in this project, when complete, will be made available as part of the PROMISE Repository of Software Engineering Databases [5]. The datasets consist of features extracted using a modified version of the open source CVSanalY [4] tool from the various CVS/RCS repositories of two open source projects with 20-year histories. The X Window System and ancillary software comprises one of the largest open source software projects in history, with hundreds of contributors. At the other extreme, the Nickle Programming Language [3] is a long-running but small project with only a few developers.

The long history of use of RCS [6] and later the RCS-based CVS [1] by open source developers suggests that extraction of software engineering features from open source repositories should be relatively easy. Unfortunately, experience gained in producing these datasets contradicts that notion. A few open source tools for analyzing CVS repository data are currently available and/or under development. Unfortunately, these analysis tools tend to be purpose-built for particular kinds of analysis, and to be problematic in other ways.

CVS makes the job of analysis more difficult by failing to preserve important kinds of transactions in the revision history. In particular, it is notorious for its inability to properly reflect file renames and directory deletions.

In addition, the important notion of *atomic commits* is not captured well by CVS. An atomic commit captures changes to a number of files of a project that comprise a single logical operation as an atomic unit. Because of its file granularity, RCS has no notion of atomic commits. While CVS is layered atop RCS, it does permit the developer to commit multiple changes in a single operation. However, the information that is needed to reconstruct that transaction is not, in general, preserved. CVS does support a commit log that is supposed to capture the information. Unfortunately, this facility is not commonly turned on: when it is, it is common for file permission problems and/or race conditions to result in either a corrupted commit log or one that is silently not updated. As a result of these problems, the derivation of transaction data in a form suitable for analysis is greatly complicated.

Finally, because CVS has historically been difficult to work with and repositories prone to corruption, projects commonly "restart"

their repository from time to time, losing revision history. In addition, the CVS "import" command, commonly used to start a CVS archive, does not preserve revision history from old CVS or RCS files by default: unless special care is taken, prior revision history will be lost. This means that work is required to track down old repositories—when it is possible at all.

Nonetheless, with persistence useful data can be abstracted from long-running open source repositories and useful analysis can be performed. The rest of this paper proceeds as follows: Section 2 describes the structure of RCS and CVS repositories (Section 2.1), the history of the specific repositories being analyzed (Section 2.2), and the tools that can be and are being used to analyze them (Section 2.3). Section 3 describes some preliminary acquisition (Section 3.1) and extraction (Section 3.2) of data from the repositories, to illustrate both the process and the possible results. Finally, Section 5 suggests some further work and draws some brief conclusions.

2. REPOSITORIES AND TOOLS

Many readers may be quite familiar with the care and feeding of open source repositories. For those who are not, a brief introduction is in order. It should be commented that a wide number of repository tools and formats are currently in use or under development: from the popular Subversion and Arch to the more esoteric such as DARCS and Monotone. Nonetheless, open source repositories with long histories tend to still be in the native format that they used at their inception: this is almost invariably CVS, usually imported from RCS.

The particular repositories used, X11 and Nickle, have their own unique characteristics that bear discussion. Further, a discussion of repository analysis tools naturally leads to a discussion of the specific tools used in this analysis.

2.1 RCS and CVS Repositories

The RCS Revision Control System is one of the earliest successful open source tools. RCS and its contemporary SCCS brought software engineering tools and practices to an open source world dominated by small, *ad hoc* projects. For better or worse, it can be argued that scaling open source project sizes and developer participation would have been impossible without the support of RCS.

The RCS repository structure is simple and functional. Repositories are manipulated by a set of small UNIX tools. Each source file in a project is treated separately by the RCS toolset: there is no relation maintained between files in the repository. The RCS tools enforce a strict locking model with file granularity. To edit a particular source file, it is *checked out* of the repository, *locking* it in the process. A locked file can be checked out only for reading by other developers. Changes to the file can be committed by *checking in* the file: this act causes the file to be unlocked and its revision history to be updated.

RCS repository files consist of the entire text of the current revision of the corresponding source file, together with a chain of reverse deltas documenting all previous revisions. Each revision is automatically numbered. Branching, experimental revision intended to explore specific alternatives, is supported by RCS. However, because the support for merging branch changes back into the main line is weak, developers use RCS branches infrequently in practice.

The CVS Concurrent Versions System is a tool originally written as a script invoking RCS for its fundamental operations. CVS provides two important enhancements to RCS. First, the non-strict revision model of CVS, while still operating at file granularity, allows multiple developers to simultaneously edit a given file in the repository. These revisions are automatically merged as needed, with support for manual resolution of conflicts. Second, and perhaps more importantly, CVS features a commit operation that automatically (for the most part) makes multiple changes to the repository simultaneously to make it correspond to the current source tree. Because CVS coordinates simultaneous changes to different files of the project, developers can concentrate on developing rather than repository operation.

Because it has historically been RCS-based, the CVS repository structure is essentially that of a collection of RCS repositories with extra metadata. The metadata mostly is stored in a special CVSROOT directory, in text files that are mostly under RCS control. Most metadata is not essential for normal operation of CVS: one can essentially drop an RCS tree into the CVS repository and start using it. By design, this property makes importing of RCS repositories trivial. It also has the important advantage that one can use RCS tools to access the CVS repository when necessary.

Unfortunately, while CVS has the ability to log atomic commit operations, it has not historically been turned on by default. In addition, the capability is not particularly robust. Since the commit log is not required by CVS for normal operation, corruption or failure of the log file may not be noticed for some time. Since RCS features no atomic commit operations and CVS fails to retain them, reconstructing them via a set of heuristics is the normal, if error-prone, approach to analysis.

Another unfortunate attribute of CVS is that it is difficult to isolate the contributions of "less-trusted" developers. While CVS supports fine-grained access control, this feature is not well known and is difficult to properly maintain: it is thus little-used. As a result, *commit access* permission to a CVS repository is often quite tightly controlled. The committer of a code revision is thus quite often not the person who actually authored that revision. This makes some kinds of repository analysis more difficult—the commit statistics of an apparently frequently-committing developer may actually represent a conglomeration of statistics from the various outside authors contributing patches to the project. These contributions are normally noted in the commit log messages, but in a way that varies from project to project and is prone to error.

2.2 X and Nickle

The X Window System and the Nickle Programming Language have obvious superficial differences. Nonetheless, these projects have some important underlying similarities. These projects were chosen for analysis for several reasons. The author has developed extensively in and for both projects. Further, the development has occurred in close cooperation with another developer, Keith Packard, who has been instrumental in both projects. As a result of this involvement, the author feels especially qualified to validate data and analysis resulting from this work.

In addition, both projects were started around the same year: 1988. They have been primarily developed in the same source language (C) and for the same platform (UNIX) using the same tools (open source compilers and development tools). As a result, they provide interesting comparisons as well as contrasts in their long-term development.

2.2.1 The X Window System

The X Window System is a premiere example of the open source development process in action. Around 1988, the MIT X Consortium released X11R1, a system comprising a display management program, a number of graphical applications to exploit the display system, and a library infrastructure supporting these applications and suitable for building further tools. The participation of work-

station vendors in the Consortium guaranteed commercial adoption, while the open source model helped to ensure interoperability and co-development among this competitive vendor community. The Consortium development team used RCS as its repository tool.

By 1994 or so, the X Consortium had become moribund, and the focus of attention for X developers had shifted to the PC-class machines that were beginning to achieve the level of computational and graphical performance to be useful as UNIX workstations. At this point, the XFree86 project effectively took over *de facto* management of the X Window System codebase.

A RCS/CVS problem noted in Section 2.1 is that it encourages a style of repository management that makes author attribution of changes difficult. This was certainly the case for the X Window System project under the X Consortium, although there were fewer outside contributors to this project in any case. Under the XFree86 project the issue of indirect contribution became more pronounced. By the time XFree86 wound down, the majority of the commits were being performed by XFree86 project leader David Dawes. The stereotypical structure of log file authorship attribution in these commits made it possible to reconstruct the authorship of individual *post facto*. However this required extra effort. More importantly, the ability to infer the actual atomic commits from the individual file changes was severely compromised by this.

In the 2001–2004 time frame, *de facto* control of X Window System migrated again, to a group led by X.org and freedesktop.org. The relevant consequence of this migration is that commit access to the X CVS repository became much more available to new developers. The result was a system that should make it much easier, going forward, to understand the development process. The X repository is also likely to migrate to tools that alleviate some of the CVS problems, perhaps Arch.

Thus, there are really three phases of X Window System development that need to be analyzed: the X Consortium phase, the XFree86 phase, and the freedesktop.org phase. Each of these analyses appears to have its own peculiar problems.

2.2.2 The Nickle Programming Language

The Nickle Programming Language started its development as a language called `ic` in the mid-1980's. The name evolved along with the system, becoming `nick` in the mid-1990's and `nickle` sometime thereafter. A full discussion of Nickle functionality is outside the scope of this paper: in brief, it is a largely conventional implementation of a vaguely C-like programming language with various fancy features.

As with X, the three phases of Nickle naming corresponded also to three different development efforts. The `ic` effort was an attempt to build a usable language for basic calculations, and for its authors to educate themselves about programming language design and development. The `nick` effort was an attempt to add more general-purpose functionality and more portability for scripting and prototyping purposes. Finally, the Nickle effort is an ongoing attempt to refine all of this into a polished system, and add some desirable-looking advanced programming language features.

The Nickle committer base has always been small, extending outside the core developers really only in the last few years. In addition, the Nickle project is quite small (currently some 50KLOC) and monolithic. As a result, Nickle reflects a different style of development than the X Window System.

2.3 Repository Analysis Tools

A recent interesting development has been that the open source community and the "traditional" software engineering community have become more aware of each other. As a result, academics and open source developers have started to build tools for performing the kind of analysis traditionally done on privately-held commercial data on open source repositories. This cooperation has been fostered by the fact that many developers, like the author, wear both academic and open source hats concurrently.

The move to open data about open projects is an important development. The inability to replicate the results of software engineering analyses on the source data has always been worrisome. Indeed, a prominent computer scientist in another field recently complained to the author at length about this problem, noting that such analyses could not be taken seriously as science. The 2004 Workshop on Mining Software Repositories [2] contains an interesting mix of reports, but perhaps the majority of them involve tools and analyses based on CVS data from open source projects. The paper by Zimmerman and Weißgerber [8] contains a nice review of work related not just to their project but to this one: it seems unnecessary to repeat that review here.

The work reported here utilizes the open source CVSAnalY tool authored by Gregorio Robles and others. CVSAnalY is a Python-based tool intended to collect and visualize development statistics about large software projects. There were several properties that recommended it for the current analysis. First, it is currently publicly freely available in source form—a surprising number of the open source tools reported on in the literature seem to be not yet released to the general public. Second, because it is written in Python, it is relatively easy to work with its source code. Third, the source has a relatively modular structure, making it easier to architect modifications. Fourth, CVSAnalY contains some nice heuristics for categorizing the "type" of files: these appear to work well and are informative. Finally, it is quite efficient: this is a critical property for analyzing the long-term high-volume X repositories. The XFree86 dataset of roughly 200K transactions takes less than an hour to generate on a standard PC workstation.

However, CVSAnalY also has some drawbacks in the current application. Most notably, the current version does not support CVS commit analysis: it does not heuristically group changes as part of a single atomic commit. This analysis should be relatively easy to add either by retrofit or as a post-pass, and may in any case be available in the next release of the tool. However, its lack currently inhibits constructive analysis of the resulting data. CVSAnalY also makes some use of the CVS metadata, inhibiting its use with the pure RCS repositories of yore. Fixing this is probably trivial, but again needs to be done to permit analysis of older data.

By default, CVSAnalY outputs its analysis to an SQL database for further processing. The author modified the tool to instead output the Attribute-Relation File Format (ARFF) files used in the PROMISE repository directly. This modification was relatively easy. The author also fixed a few buglets in CVSAnalY to prepare it for use.

3. DATA CAPTURE AND ANALYSIS

With all of the above in mind, the capture and analysis of the repository data appears straightforward. There are really only three basic steps: acquire copies of the needed repositories, run the modified CVSAnalY tool on the repositories to extract the relevant data, and then run analysis tools such as Weka [7] on the data to understand what it is saying. The devil, as usual, is in the details, and sometimes even in the broader picture.

3.1 Data Acquisition

For open source projects that the author has been intimately involved in since their inception, one would expect that acquiring copies of the repositories would be a snap. Unfortunately 20 years

Table 1: Repository summaries

Codebase	Changes	Authors	Lines+	Lines-
XFree86	141K	21	5.85M	2.4M
X.org	133K	42	5.1M	1.9M
Nickle	2972	6	71.7K	34.6K

is a long time, and a lot of bad things can happen to data. The real heart of the data acquisition problem lies in the earlier observation (Section 2.1) that CVS encourages a style in which repositories are periodically re-created.

For X, this happened with each organizational change: the new organization imported the repository in such a way that the old revision history was essentially lost. Thus, ultimate copies of the X Consortium and XFree86 repositories had to be located. This proved easy for XFree86. The old repository is still nominally active, and can be easily downloaded.

For the Consortium code, the situation is more problematic. Unfortunately, because of legal issues with the X Consortium, the XFree86 developers were unable to acquire the Consortium RCS repository. Thus, they started afresh with a new repository. Attempts to locate archives of the original Consortium RCS repository have so far been unsuccessful. A current focus of effort is to restore old ClearCase VOB archives that may contain RCS transaction information from that period. Failing this, the early revision history of X may be permanently lost.

For Nickle, the repository re-creation happened for each new development phase. The original `ic` RCS repository is available. While versions of the `nick` source code have been located, the underlying CVS repository appears to be permanently lost. This repository resided on a machine that author Keith Packard had access to when he worked at NCD, a now-defunct manufacturer of X Window System based hardware. Apparently, no archive was made upon Packard's departure from the organization, and the repository was inadvertently not incorporated into the Nickle repository due to the problems with CVS discussed in Section 1. The Nickle repository, however, is in good shape.

3.2 Data Extraction

Of course, having a repository is useless unless the analysis tools can be made to operate on it. As mentioned above (Section 2.3), CVSAnalY does not currently quite support RCS-only repositories. Add this to the data acquisition problem, and only three repositories remain to be analyzed: the current Nickle repository stretching back about 6 years, the X.org repository stretching back just a few years, and the XFree86 repository.

These three repositories were analyzed using CVSAnalY: the preliminary results are available online as `.arff` files. Table 1 summarizes these datasets. The Lines+ and Lines- columns indicate the total number of lines added to and deleted from the repository over the analysis period. These totals exclude data for which CVSAnalY infers type "image" or "multimedia": this data appears to be seriously over-counted by CVSAnalY. The X data also excludes a very few adds that appear to be corrupted by a CVSAnalY bug. The XFree86 summary data excludes commits from 2003 or later, since these are assumed to be largely incorporations of changes from the X.org codebase.

One interesting fact that emerges immediately is that the X.org repository has already managed to accumulate almost as many file deltas as XFree86 did over its entire useful life. This suggests that the desire to achieve more rapid development is in fact being met. Note that it is harder to draw conclusions about greater authorship, since CVSAnalY is currently unable to acquire the actual author of commits, as discussed previously (Section 2.2.1).

Another interesting observation is that, over the long haul, additions appear to outnumber deletions at the rate of about 2::1 for all three repositories. It would be interesting to compare this with the rate from other long-term open source projects.

For each repository, CVSAnalY extracts 7 salient attributes and heuristically synthesizes two more. These 9 attributes are

1. **Inferred file type.** A heuristic is used that attempts to guess the general classification of the file being altered from its filename. There are 9 possible classification categories: *documentation*, *images*, *i18n*, *ui*, *multimedia*, *code*, *build*, *develdoc*, and *unknown*. For the datasets reported here, only a small fraction (10–20%) of the files are classified *unknown* by the heuristic. Informal random sampling revealed fairly high subjective accuracy for the positive classifications.

2. **File pathname.** The relative directory and name of the file affected by the change.

3. **Revision.** The RCS revision number produced by the change.

4. **Author ID.** A numeric encoding of the author field of the change. As discussed previously, this is actually the committer, and may commonly fail to reflect the true author.

5. **Lines added.** The number of lines added by the change. An apparent bug in CVSAnalY caused a very few changes in the X datasets to be flagged with a negative value in this field. These anomalous change records were ignored in the statistics reported here.

6. **Lines removed.** The number of lines deleted by the change. Note that there is no real way to capture "lines changed" in current revision control software. Thus, each line changed is logged as a line added and a line deleted. This is arguably wrong: a heuristic should be used to group adjacent insertions and deletions of a single change into "changed lines" to avoid over-counting and other confusion.

7. **File "in Attic".** This boolean flag indicates that the file has been explicitly deleted from the repository. An artifact of the inability to rename a file in CVS and the concomitant common work-around of notionally deleting the file and recreating it under a new name is that many files are misleadingly marked as deleted.

8. **Inferred non-author commit.** CVSAnalY analyzes commit log messages using a fairly weak heuristic to try to discover whether an author other than the committer of a change is mentioned in the change log entry. While the heuristic used probably massively undercounts the frequency of non-author changes, it does occasionally provide an indication that this has happened.

9. **Commit date.** The date the file was committed. One must assume, in the absence of evidence as to the workings of old versions of CVS, that the date is GMT. This is good for comparing changes from geographically diverse committers, but less good for inferring things like whether daytime or nighttime changes are more common.

For summary statistical information about each of these attributes for each of the data sets reported here, please see the headers of the relevant PROMISE Repository data files.

4. USING THE DATA

The datasets as currently collected are actually quite weak. In particular, the lack of code quality and/or defect data correlated with changes is problematic. The inability to determine the atomic commit associated with a change is also a problem. However, there are still interesting things to be discovered by analyzing this data.

A particular focus of software engineering metrics has been on measuring productivity. This is a relatively easy thing to do in the traditional software development cycle. Commercial projects tend to be developed over months or at most a few years. During the development phase, control and measurement of productivity in many organizations is quite detailed. The maintenance phase, however, is another matter. Studies of software maintenance productivity are made more difficult in commercial settings by the relative lack of attention payed to maintenance in many organizations.

When maintenance and development are integrated, as they tend to be in long-timescale open source projects such as those reported here, the productivity questions are at once more interesting and more important. Can we find automatically measurable characteristics distinguishing maintenance and development activities when these activities are intermingled in an open-source project? Can we use this sort of information to change software development practices and improve productivity? One might hypothesize a positive answer to these questions without embarrassment.

For example, consider development around release dates for open source projects that release well-labeled major versions from time to time. Packard hypothesizes that these release dates become focal points for development. Under this hypothesis, one would expect to see an increasing frequency of large changes up to the release date, as major features are incorporated. Once the release date has passed, a decreasing frequency of smaller changes should occur, as the user defect reports so important in open source development trigger restorative maintenance activities.

This hypothesis suggests that machine learning on or statistical analysis of the datasets reported here with change size and frequency as dependent variables would be an interesting experiment. This experiment suggests other such experiments. How does the productivity of individual developers, again measured in terms of change size and frequency, evolve over the course of their tenure with an open source project? What directories in an open source project are the focus of frequent intense change activity? The answers to these kinds of questions would seem to have potentially important implications in improving the software productivity, not just of closed-source development, but of open-source development also.

5. FUTURE WORK

In the near term, further attempts will be made to completely acquire the X repository. In addition, the prospects of acquiring other long-timescale open source repositories will be explored. GCC, for example, seems a promising candidate. A particular focus should be on finding such a repository that can be correlated reliably with a source of defect data.

The authors are currently working on using the data already acquired to develop the hypotheses and suggested experiments of Section 4.

The longitudinal study of long-standing open source codebases by automated repository data extraction promises to give insights into code evolution and maintenance. Researchers are taking the first steps toward achieving this promise.

Acknowledgments

Thanks to Tim Menzies for inspiration and for useful discussions of the project. Special thanks to Gregorio Robles for making CVSAnalY available on an open source basis. Thanks to Keith Packard and Jim Gettys for help in obtaining and understanding X repository data. Thanks also to Packard for reviewing drafts of this paper and suggesting some of the analysis proposed in Section 4. Finally, thanks to the PROMISE Repository folks for providing a forum for this sort of work.

Availability

The data described in this paper, as well as the modified version of CVSAnalY used to produce it, is available via the PROMISE Software Engineering Repository at `http://promise.site.uottawa.ca/SERepository/`.

6. REFERENCES

[1] K. Fogel and M. Bar. *Open Source Development with CVS*. Coriolis, 2001.

[2] A. E. Hassan, R. C. Holt, and A. Mockus, editors. *Proceedings of the 1st International Workshop on Mining Software Repositories*, Edinburgh, Scotland, May 2004.

[3] B. Massey and K. Packard. Nickle: Language principles and pragmatics. In *Proc. 2001 Usenix Annual Technical Conference, Freenix Track*, Boston, MA, June 2001. URL `http://www.nickle.org/usenix-nickle.pdf`.

[4] G. Robles, S. Koch, and J. González-Barahona. Remote analysis and measurement of Libre software systems by means of the CVSanalY tool. In *ICSE 2004—Proceedings of the Second International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS '04)*, pages 51–55, Edinburgh, Scotland, 2004.

[5] J. Sayyad Shirabad and T. Menzies. The PROMISE repository of software engineering databases. School of Information Technology and Engineering, University of Ottawa, Canada, 2005.

[6] W. F. Tichy. RCS – A system for version control. *Software – Practice and Experience*, 15(7):637–654, July 1985.

[7] I. H. Witten and E. Frank. *Weka: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.

[8] T. Zimmermann and P. Weißgerber. Preprocessing cvs data for fine-grained analysis. In Hassan et al. [2].