On Mapping Releases to Commits in Open Source Systems

Joseph F. Shobe, Md Yasser Karim, Motahareh Bahrami Zanjani, Huzefa Kagdi
Department of Electrical Engineering and Computer Science
Wichita State University
{jfshobe, mxkarim, mxbahramizanjani, huzefa.kagdi}@wichita.edu

ABSTRACT

The paper presents an empirical study on the release naming and structure in three open source projects: Google Chrome, GNU gcc, and Subversion. Their commonality and variability are discussed. An approach is developed that establishes the mapping from a particular release (major or minor) to the specific earliest and latest revisions, i.e., a commit window of a release, in the source control repository. For example, the major release 25.0 in Chrome is mapped to the earliest revision 157687 and latest revision 165096 in the trunk. This mapping between releases and commits would facilitate a systematic choice of history in units of the project evolution scale (i.e., commits that constitute a software release). A projected application is in forming a training set for a source-code change prediction model, e.g., using the association rule mining or machine learning techniques, commits from the source code history are needed.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

Mining Software Repositories, Software Releases, Commit History, Empirical Studies

1. INTRODUCTION

One common thread of investigation in the software maintenance and evolution community is improving software (e.g., bug or change) predictive models. Oftentimes, these models rely on the premise that an event, e.g., a defect or change in a source code entity, which occurred in the past is likely to reoccur in the future[1][2]. A number of previous studies in the literature used time periods in calendar units of day, month, or year (e.g., 2 or 6 months) to gather events in the training set formation of a desired prediction model[1][2]. The rationale for a particularly chosen slice of history ranges from the observation of a specific development or maintenance activity to arbitrary.

We propose an approach that would facilitate the formation of training sets in release units of the software project. Releases form organic units of the software evolution time line to stakeholders ranging from end users to developers. The project release names/numbers (e.g., major or minor release) are typically an indicator of the magnitude of changes and are primarily reference points in the production environment. For example, a user reports a bug found in a particular release of a software system (e.g., *gcc* 4.7.0). Another scenario is a feature request to be added to the next release of a software system. In case of a certain source-code change prediction model, the training set includes commits, i.e., changes to source code entities, which are typically stored in version control systems. The explicit mapping between a specific release number and Subversion commits forming this release including the repository location of these commits (e.g., branch or trunk) is valuable; however, oftentimes not directly available from the project. One could infer release to commit mappings from release dates, which are often available from the project web site. Unfortunately, this approach provides a coarse approximation of the commit range at best, possibly including irrelevant or excluding relevant commits, and no precise location of these commits (e.g., trunk or a specific branch).

Our approach establishes mapping from the project release numbers (macro evaluation, which the broader user community sees) to the specifics of the changes in the repositories (micro evolution, needs of the developers to support the change task). It maps project release numbers (e.g., *gcc* 4.7.0) to Subversion commit revisions (e.g., revision 184777 to 194296) and their location in the repository (e.g within the branch gcc-4_7-branch). To form a basis for our approach, we conducted an examination of three open source projects: *Chrome*, *gcc*, and *Subversion*. We found that the release to revision mapping is neither explicitly documented nor is straightforward to infer. It requires a systematic engineering approach augmented with human intervention. Furthermore, the release naming conventions and classifications (e.g., major and minor releases) vary across projects. We did not find any explicit description of release to commit mappings in the previous studies on predictive models reported in the literature.

2. EXPLORATORY STUDY ON RELEASE SCHEMA

Subversion is a centralized version control system which is commonly used to maintain the source code repository of a project. A single Subversion commit is a transaction of changed files bundled together for inclusion in the repository. During the software development life cycle, individual files to be modified or deleted (new files are handled during the commit) are checked out from the trunk or branch of the source code repository and then later committed back to the same trunk or branch where a unique identifier (i.e. revision number) is assigned. Eventually, committed transactions are grouped together to form a *release.*

A *major* release typically indicates substantial changes in the software, such as new functionality, new hardware platform, or perhaps a redesign of the presentation approach.

A *minor* release typically indicates fixes to the code base without introducing any substantial change to functionality.

Sequence-based versioning schemes with three and four elements (such as 3.4.1 and 4.2.1.3) are popular approaches for identifying the name of a major/minor release for a project. A very common practice is to use the first part for identifying the major release and the second for identifying the minor release within a major release. Thus, the release number 4.2.1.3 has a major release of 4 and a minor release of 2. Another, less common practice, is to use the first two parts for identifying the major release and the remaining parts for identifying the minor release. Thus, the release number 3.4.1 has a major release of 3.4 and a minor release of 1.

Three open source projects, *Chromium* (*Chrome* for short), *gcc*, and *Subversion*, were chosen to understand the breakdown of their release schemas.

2.1 Chrome

The *Chrome* website has several documents regarding the development and release of the project. One document[1] shows a time line of some of the latest releases (see Table 1) and their corresponding dates. The list is incomplete because it does not show releases earlier than 17; however, it does serve to establish the definition of the release name and how often releases occur (between one and two months). In another document[2] found on their web site, the release-numbering scheme is described in detail. Each release consists of a four-part sequence (e.g., 25.0.1331.0). The first and second parts identify the major and minor release number (e.g., 25 and 0). The third identifies the build number (e.g., 1331) and fourth identifies the patch number within a build (e.g., 0).

Table 1: A Subset Of Chrome Releases

Release	Week of Branch Point
23	Sept 17th, 2012
24	Oct 29th, 2012
25	Dec 17th, 2012

Each release of *Chrome* is stored within the releases directory[3] and details about each release directory include the directory name, latest revision number, its age from today, the author and the last entry in the log. Upon examination of the project's repository, the versioning for a major release does indeed follow the pattern described in the project documentation. That is, the first part of the release sequence describes the major while the second part describes minor. Out of twenty-five major releases, only two of them, 4 and 17, included one minor release. Development for *Chrome* occurs along the trunk of the repository. When a release (more than just a patch) is desired, a build (branch) is created by taking a snapshot of the trunk or another branch and the newly created build number is used as the directory name[4]. In addition, a release (tag) is created, which is neither a snapshot of the trunk nor branch. Rather, it includes a couple of new files, one of which is a dependency file identifying all the files included in the release and their location in the trunk, a branch, or some external system. Smaller changes to a major release are additional builds, each with a new release signature (due to the increasing build number).

[1]http://www.chromium.org/developers/calendar
[2]http://www.chromium.org/releases/version-numbers
[3]http://src.chromium.org/viewvc/chrome/releases/

2.2 Gcc

The *gcc* web site also has several documents. One such document[5] describes the development and release methodologies; past, present, and future development, branch points, and releases in a tree-like manner. Another document[6] shows a timeline of releases and their dates.

Table 2: A Subset Of Gcc Releases

Release	Release Date
Gcc 4.7.2	Sept 20, 2012
Gcc 4.5.4	July 2,2012
1.3	June 10, 1987

Their release-numbering scheme is more traditional, consisting of three parts. By observation and reason, one can conclude the first part is the major, second part is the minor, and third part is a patch or fix. For this project, the focus is along the major/minor releases (e.g., 4.6, 4.7). The web site also includes the ability to view the project's source code repository[7]. Details about each directory include the directory name, latest revision number, its age from today, the author, and the last entry in the log. Development for gcc occurs along the trunk of the repository. When a release (more than just a patch) is desired, a branch is created by taking a snapshot of the trunk or another branch and storing it in the branches directory.[8] using the newly created branch as the directory name. In addition, a tag is created by taking a snapshot of the branch and storing it in the tags directory.[9] using the newly created tag as the directory name. Patch releases occur within an existing branch and generate a new release signature, but not a new branch.

Beginning with release 3.0, the major, minor, and patch level values translate rather easily into branch and tag names listed within the branches and tags directories of the repository. Branches are prefixed with the string "gcc-"followed by the major, minor of the release, and finally a postfix string of "-branch"(e.g., 4.7 has a branch name of "gcc-4_7-branch"). Each release tag is prefixed with the string "gcc_" followed by the major, minor, patch level of the release, and finally a postfix string of "_release" (e.g., 4.7.2 has a tag name of

[4]http://src.chromium.org/viewvc/chrome/branches/
[5]http://gcc.gnu.org/develop.html
[6]http://gcc.gnu.org/releases.html
[7]http://gcc.gnu.org/viewvc/gcc
[8]http://gcc.gnu.org/viewcvs/gcc/branches/
[9]http://gcc.gnu.org/viewcvs/gcc/tags/

"gcc_4_7_2_release").

2.3 Subversion

The web site for *Subversion* includes a document [10] describing the time line of all releases (see Table 3) and another document [11] outlining specific details about the making of a release, including details of the version numbering scheme. Their scheme uses a three-part numbering system. The first part identifies the major, the second part the minor, and the last part is for identifying patches or bug-fixes. The web site also includes the ability to view the project's source code repository [12]. Of interest in this directory are the tags, branches and trunk directories.

Our work focuses along the major / minor releases (e.g., 1.6, 1.7). These too translate into the tag names listed in the tags directory of the repository for each of the releases. Each release is simply the major, minor, patch level of the release (e.g., release 1.6.1 has a tag name of "1.6.1"). Each release tag is a snapshot of a branch from the branches directory and corresponds to the major and minor release. Branches are almost the same. Each begins with the major, minor of the release, and ends with a string of ".x" (e.g., branch 1.6 has a name of "1.6.x"). Each branch is usually a copy of the trunk at the time the branch was created; however, a couple of exceptions occurred with respect to the 1.0 release. First, the branch 1.0.x came from another branch that no longer exists in the directory instead of coming from the trunk. Second, a few of the release tags came from a branch name that includes the patch level (e.g., 1.0.3) instead of the expected branch 1.0.x.

3. MAPPING RELEASES TO REVISIONS

In this section we use our understanding of release schemas to map major and minor releases of three open source projects to identify the start (earliest) and end (latest) commit revisions from both the branch and trunk for each release.

3.1 Extracting Commit History

Using Subversion's command line tool (svn), we can access each project's repository and list a directory as well as retrieve logs for a single commit (by revision number) or range of commits (range of revision numbers). Thus far, we only determined the names of the releases and their location in the directories. We are yet to determine the starting (earliest) revision number for each of the releases. It is important to note this information is not available and must be constructed manually through examination of commit logs.

3.2 Origin of Releases and Branches

By using the above technique, the latest and earliest revision numbers can be located for each of the major and minor releases of interest as well as each branch because each has its own directory. However, where each of the releases and branches came from still needs to be determined. Did the release come from a branch or the trunk? Did the branch come from another branch or the trunk? The commit logs do include a list of files and directories that were added, modified, deleted, or replaced. These logs will be helpful in deriving the answer to these questions.

For *Chrome*, the logs for each release only show the addition of a couple of files (using the *svn -v* flag includes more verbose information such as details of the directories/files affected), so there is nothing in the logs to indicate the source of the release. Thus, the release is not a snapshot of its source. However, the release directory itself does include the build number. As stated earlier, the build number is also the name of the branch in the branches directory, so the source of the release can be derived as always coming from the branch. For *gcc* and *Subversion*, one must examine the logs for each release within the tags directory to identify its source. Each release's earliest log includes a line entry showing the addition of the directory name itself. As tags are a snapshot of a branch or trunk, the line entry reflects the name of the entity (file or directory), the source of where it came from (branch or trunk) and what the latest revision number in the source was at the time of the snapshot (copy). For these two projects, the data collected shows release tags come from branches (with a couple of exceptions) and thus, the specific branch and revision number can be identified by examining the line entry in the logs. One exception to this rule are the tags for older releases that no longer supported, which shows they came from the trunk or from a branch that no longer exists, but these were due to the migration of the repository. Another exception, specific to *Subversion*, showed their 1.7 alpha releases came from the trunk.

This process of identifying the location of releases and branches is manual and takes a lot of times. To build an automated tool for locating positions , all the projects should have some common branching techniques. From above discussion, it was clear that *Chrome* had different branching scheme than *gcc* and *Subversion*. Besides we had found that in *Subversion* there were some extra branching for migration process of version control system. These obstecles were the main challenges for us to build an automated tool. Our future work also focus on this track for further development.

3.3 Release to Commit Mapping

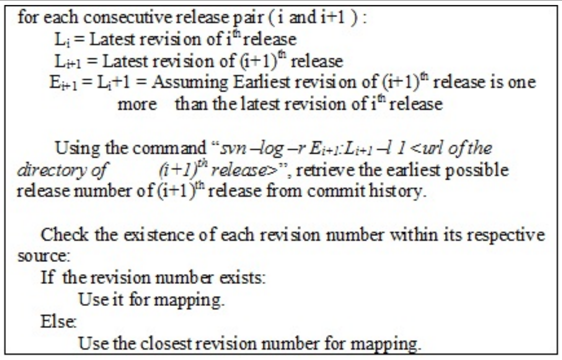
Development for the next release, be it a major or minor, occurs within the trunk. When a new release is desired, a snapshot (copy) of the trunk is taken to create a new branch and then (as seen for gcc and Subversion) a snapshot of the new branch is taken to create a new release, thus branches come from the trunk and releases come from the branch. When a new patch release is desired, a snapshot of the branch is taken, thus patch releases come from an existing branch. With this understanding, the svn tool can be used to pull logs to identify the starting (or earliest) revision for a release or branch. The logs of interest are those that exist between the latest revisions of two consecutive releases or branches. Specifically, the algorithm in Figure 1 is used.

For example, using gcc's branches 4.4(i) and 4.5(i+1), the L_i value for branch 4.4 was pre-determined to be '45121', therefore E_{i+1} has a value of '145122'. The L_{i+1} value for branch 4.5 was also pre-determined to be '157987'. As a result, the output of "*svn log -r 145122:157987 ..*" yields a log identifying the earliest revision of branch 4.5 as '145124' with a date of '03-27-2009'. Using this method, we have mapped releases to revision numbers of version control system as presented in Table 4.

[10]http://subversion.apache.org/docs/release-notes/release-history.html
[11]http://subversion.apache.org/docs/community-guide/releasing.html
[12]http://svn.apache.org/viewvc/subversion

Table 3: A Subset of Release to Revision Mappings

Version	Release	Latest Revision	Earliest Revision	Branch	Latest Revision	Earliest Revision	Trunk	Latest Revision	Earliest Revision
Chrome									
24.0	24.0.1272.0	157735	157734	1272	165396	157728	trunk	157686	150761
25.0	25.0.1313.0	165136	165135	1313	165127	165127	trunk	165096	157687
Gcc									
4.6.0	gcc_4_6_0_release	171513	171513	gcc-4_6-branch	171512	170935	trunk	170934	157990
4.7.0	gcc_4_7_0_release	185675	185675	gcc-4_7-branch	185674	184777	trunk	186776	170936
Subversion									
1.6.0	1.6.0	876759	876759	1.6.x	876724	875962	trunk	875961	869157
1.7.0	1.7.0	1181106	1181106	1.7.x	1176459	1145993	trunk	1145982	875964



```
for each consecutive release pair ( i and i+1 ) :
      L_i = Latest revision of i^th release
      L_{i+1} = Latest revision of (i+1)^th release
     E_{i+1} = L_i+1 = Assuming Earliest revision of (i+1)^th release is one
               more   than the latest revision of i^th release

      Using the command "svn –log –r E_{i+1}:L_{i+1} –l 1 <url of the
directory of        (i+1)^th release>", retrieve the earliest possible
release number of (i+1)^th release from commit history.

    Check the existence of each revision number within its respective
source:
    If the revision number exists:
          Use it for mapping.
    Else:
          Use the closest revision number for mapping.
```

Figure 1: Mapping Algorithm

4. RELATED WORK

There are a number of research efforts on the nature of commits. Some researchers have tried to relate commits to certain activities, e.g., large commits are more likely to be originated from code management activities, while small ones are related to development activities [3]. Alali, et al. [4] examined version histories of nine open source software system to uncover trends and characteristic of what would typical commits look like. Gall et al. [5] used the release information such as version numbers of programs, modules, and subsystems together with change reports to discover common change behavior such as change patterns. Gall et al. [6] studied 20 different releases of a system with different major and minor releases. Saliu et al. [7] proposed a new release planning framework that considers the effect of existing system characteristics on release planning decisions.

From the above discussion, it can be seen that none of the approaches traced releases to commits. Our approach provides this mapping, which has projected applications for tasks such as change impact analysis, change prediction, and detecting logical coupling.

5. CONCLUSIONS AND FUTURE WORK

We studied three open source systems and found that there are differences in their release schemes. There are noticeable variations in their naming convention, repository structures, and the mapping information to source code commits. We presented an approach to trace the specific release to the commit range (first and last commits) in the source code repository. The desired applications of this release to commit mapping include the formation of effective training sets and change predictive models. Our immediate goal is to empirically compare the time-based selection of commits and our approach with regard to prediction tasks (e,g., change and bug) and their accuracies.

6. ACKNOWLEDGMENTS

This work is supported in part by the NSF CCF-1156401 grant. Any opinions, findings and conclusions expressed herein are those of the authors and do not necessarily reflect those of the sponsors

7. REFERENCES

[1] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *Software Engineering, IEEE Transactions on*, 31(6):429–445, 2005.

[2] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, 2007.

[3] L.P. Hattori and M. Lanza. On the nature of commits. In *Automated Software Engineering - Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on*, pages 63–71, 2008.

[4] A. Alali, H. Kagdi, and J.I. Maletic. What's a typical commit? a characterization of open source software repositories. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 182–191, 2008.

[5] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 190–198, 1998.

[6] H. Gall, M. Jazayeri, R.R. Klosch, and G. Trausmuth. Software evolution observations based on product release history. In *Software Maintenance, 1997. Proceedings., International Conference on*, pages 160–166, 1997.

[7] O. Saliu and G. Ruhe. Supporting software release planning decisions for evolving systems. In *Software Engineering Workshop, 2005. 29th Annual IEEE/NASA*, pages 14–26, 2005.