Detecting Patch Submission and Acceptance in OSS Projects

Christian Bird, Alex Gourley, Prem Devanbu
Dept. of Computer Science
UC Davis
Davis, CA 95616,USA
cabird,acgourley,devanbu@ucdavis.edu

Abstract

The success of open source software (OSS) is completely dependent on the work of volunteers who contribute their time and talents. The submission of patches is the major way that participants outside of the core group of developers make contributions. We argue that the process of patch submission and acceptance into the codebase is an important piece of the open source puzzle and that the use of patch-related data can be helpful in understanding how OSS projects work. We present our methods in identifying the submission and acceptance of patches and give results and evaluation in applying these methods to the Apache webserver, Python interpreter, Postgres SQL database, and (with limitations) MySQL database projects. In addition, we present valuable ways in which this data has been and can be used.

1 Introduction

One of the primary tenets of the open source philosophy is that anyone can decide to contribute to a particular project in any way that they want. The success of OSS projects is highly dependent on a stream of newcomers who are able and willing to contribute in a number of ways such as writing documentation, fixing bugs, adding new features, sharing technical expertise, providing support to users, etc. While each OSS project has a core group of developers with write access to the source code (and in some cases documentation) within it's repository, newcomers without this privilege can also make contributions. These contributions are primarily made by submitting patches on project mailing lists. In our investigations, the mailing list participant pool is usually one to two orders of magnitude larger then the inner circle of developers.

However, only some of the mailing list participants actually submit work-gifts in the form of patches. Most developers are drawn from this smaller group. Therefore, the patch submission and acceptance process is critical to OSS communities and worthy of study. We present a method of collecting patch submission and acceptance data that has largely been overlooked until now. This mined data can give us finer grained insights into OSS communities (which have largely been divided into the roles of developers, dev mailing list participants, and users), give more information about the files contained in project file repositories, and tell us more about developers themselves by examining their submissions prior to becoming developers and their reviews afterwards.

We have developed a method to both detect patch submissions on the project mailing lists and determine if the submitted patch was applied to the code base within the project repository. In this paper we discuss some of the issues faced in datamining patches in an open source project and present our methods for overcoming them. We also present our findings and a preliminary evaluation from applying these methods to the Apache, Python, PostgreSQL, and MySQL projects. In addition, we discuss ways in which this patch data has been used and can be used in the future.

2 Related Work

This work is inspired by two studies of OSS projects. Nicolas Ducheneaut performed an ethnographic study on the Python project which emphasized the importance of patches in the development and social process [2]. Von Krogh *et al* examined joining scripts (among other things) for newcomers and the processes by which they made contributions to the Freenet project [4]. In addition, there is a large body of literature which supports the notion that patch submission is a critical aspect in open source software.

3 Methodology

The process of detecting patch submissions and acceptances is part of a much larger project at UC Davis that aims to mine many forms of data from various OSS projects and use that data to understand how these projects work. We already have many of the tools in place for this form of detection and analysis. For each project under study, we have downloaded and analyzed the complete source code repository and the developer mailing lists. The results of the analysis, stored in a database system, contains information such as who made changes to what files in the repository or what messages were sent on a particular date. With the full text of every message sent and every file ever checked into the repository, we are ready to begin the process of detecting submitted patches and determining which of them were accepted and applied. In the OSS world, a patch file usually represents a number of changes to be applied to multiple files within a codebase. For each file, there is a series of contiguous changes called patch hunks. For our purposes, we divide these multi-file patches into individual patches, one per file needing modification. Thus, if a contributor posts a message that contains a patch modifying two files, we treat these as two separate patches. Without loss of information this allows us to examine patch acceptance at a finer level of granularity.

3.1 Patch Submission Detection

The developer mailing list(s) is , in most projects, the prescribed medium by which patches are submitted for review and application to the repository. Therefore, the first step of detecting patch submissions and extracting the bodies of those patches for use later includes analysis of the full text of the messages posted on this mailing list.

Patches are created by running the `diff` utility on original and modified versions of the same file or files. Thus, although there multiple formats for patches (e.g. unified, context, etc), the number of forms is limited and easily recognizable.

For purposes of illustration, below is a "patch offering" from mailing list participant Mark Bixby suggesting a modification to the "configure" file in apache 1.3.10 after testing a recent change on the MPE/iX architecture.

```
From: Mark Bixby <mark_bixby@hp.com>
Subject: RE: Test the baby...
Date: 2000-01-17 14:34:28-08

Looks good on MPE/iX, except for some minor configure breakage.
Could somebody please apply this patch for me?

Thanks!
- Mark B.

--- apache_1.3.10/configure     Tue Jan 11 11:47:42 2000
+++ apache_1.3.10_m/configure   Mon Jan 17 13:55:58 2000
@@ -339,6 +339,10 @@
         iflags_program="${iflags_program} -e .exe"
         iflags_core="${iflags_core} -e .exe"
         ;;
+    *MPE/iX* )
+       default_layout="Apache"
+       iflags_program="-m 755"
+       ;;
    *)
        default_layout="Apache"
@@ -357,9 +361,6 @@
             set -- --with-layout="$default_layout" "$@"
         fi
         ;;
-    *MPE/iX* )
-       iflags_program="-m 755"
-       ;;
 esac
```

We use a series of regular expressions to detect any known forms of headers in the bodies of all messages from the mailing lists. The headers indicate the form of the patch, the name of the file being patched, and the path to the file in the repository (though the last is not always accurate).

We dealt with two issues when detecting and extracting patches from the messages. First, since the body of the patch is embedded within the normal text of the message, it is difficult to determine where the patch actually ends. This is partially overcome by using the line counts contained within the headers for each hunk of the patch. However, this is complicated by the second issue, which is that some email clients automatically wrap long lines of text (which often occur in source code). We therefore use a hand-tuned heuristic based on the content of each line (such as whether the first character is whitespace or not) to determine where the patch ends and the normal email communication begins again. Random sampling indicates that this technique incorrectly marks the end of a patch in an email less than 5% of the time.

Once the patches have been detected and extracted from the messages, we store the body of the patch along with author and file information in our database.

3.2 Finding Patch Applications

We're also interested in which patches were actually accepted into the codebase. At a high level, this process seems relatively trivial. One simply needs to determine which file in the repository the patch references and checking the patch against every version of that file. This ends up being more difficult than it sounds. We have found that the directory path in patch headers are often inaccurate. Therefore, in order to determine which file the patch may apply to, we search the entire repository for any file with the same name. In addition, in order to reduce computational time, we constrain the versions of files that we test to those that exist in the time interval from one day before to 80 days after the patch submission. After testing using without this limitation, we found that this interval safely finds nearly all successful patch applications (common sense and experience also indicate that patches aren't applied much earlier or later than time of submission).

Checking a particular version of a file to see if it appears that a patch has been applied is also somewhat difficult. It's possible to use the `patch -R` command, which attempts to apply the patch in reverse. The problem with this approach is that patched files are sometimes modified prior to being committed to the repository. If a developer applies a submitted patch and then moves a curly brace from the end of one line to the beginning of the next, the original patch will not reverse apply and this approach will yield a false negative. In this case the problem is that patches are line based. Other issues that we encountered causing false negatives were comments being added, removed, or modified, and variables being renamed prior to committing patched files.

In an effort to improve the recall of our patch application detection process, we wrote our own tool to deal with these problems and other issues that may be encountered in the future. To address the problem of line based changes (such as the moved curly brace above), our tool reads both the patch and candidate file and produces two sequences of tokens. We use specialized scanners for files depending on the programming language used (determined by file extension) and also a generic scanner for files containing natural text or unrecognized languages. Due to the object oriented nature of our tool, it is quite easy to extend the generic scanner to recognize new languages by simply specifying the keywords, comments, identifiers, symbols, etc. with extended regular expressions.

Once the candidate file and patch have been tokenized, we utilize various matching techniques to determine if the patch token sequence appears in the candidate file token sequence. To deal with comment addition, deletion, and modification, we attempt to match the patch token sequence against the file token sequence both including and not including comments. In some cases, we found patches that consisted solely of a modification to a comment in order to clarify code. Our tool is able to recognize cases such as this and act accordingly. We also account for other problems such as tokenizing correctly when patches begin or end within a multi-line comment.

Identifier renaming is a common modification to patched files prior to committing. Many projects have naming conventions that well-meaning newcomers may not be aware of. To mitigate this problem, we check to see if the patch token sequence appears in the candidate file modulo identifier-renaming. Below is an example of such an occurrence from our own data. In this case, the type `ap_bucket_brigade` in the patch has been renamed to `apr_bucket_brigade` in the file after patch application but prior to commit. The first section of code is from the body of a submitting patch while the second section represents portions taken from mod_cgi.c itself.

```
+typedef enum {RUN_AS_SSI, RUN_AS_CGI} prog_types;
+typedef struct {
+    prog_types          prog_type;
+    ap_bucket_brigade **bb;
+} exec_info;
+
 /* KLUDGE --- for back-combatibility, we don't have to check ExecCGI
...
+    const char   **argv;
+    ap_bucket_brigade *bcgi;
+    ap_bucket *b;
+
-----------------------------------------------------------------
typedef enum {RUN_AS_SSI, RUN_AS_CGI} prog_types;
typedef struct {
    prog_types           prog_type;
    apr_bucket_brigade **bb;
} exec_info;

/* KLUDGE --- for back-combatibility, we don't have to check ExecCGI
...
    const char   **argv;
    apr_bucket_brigade *bcgi;
    apr_bucket *b;
```

We found that in some cases, the patch itself only applied a renaming of variables (such as when platform specific `#define`'s in GCC have changed). Our tool also checks for this and requires exact identifier matches in these cases.

We examine each hunk of the patch in turn and check to see if it was applied to the corresponding file. We use a tuneable threshold for the proportion of hunks that must be applied for the patch to be considered successfully accepted. For our purposes, we set this value to around 3/4[1]. If a successful application is detected, the corresponding database entry for the patch is updated with the matching file location and version information for use in later analysis.

4 Results

We have tested our patch mining process on the Apache webserver, the Python interpreter, and the PostgreSQL and MySQL database systems. In the case of MySQL, we only gathered submission data because we were unable to check out versions of their files due to their use of a proprietary source code management tool (BitKeeper). Table 1 shows the results of our patch mining for these projects. The values indicate the total number of patch submissions, number of submissions by non-developers, total number of acceptances, number accepted from non-developers, and distinct number of people who made submissions and had accepted patches.

4.1 Evaluation

In order to evaluate our patch mining process we need to measure both the recall and precision of our results [3]. Measuring precision requires testing for false

[1] This value is more of a "gut feel" than anything else. We wanted a patch to be mostly accepted for us to indicate it as so.

	Apache	Python	Postgres	MySQL
Patches Submitted	4267	644	1209	185
by non-devs	2101	315	856	137
Patches Accepted	1087	173	591	NA
from non-devs	448	69	407	NA
ppl who submitted	253	106	172	47
ppl with accepted	106	44	92	NA

Table 1. Results of using our tool on projects studied.

positives, which turns out to be quite easy. We can examine a patch and the file that the patch was applied to according to our tool and a quick inspection can usually determine if the two do in fact match. A random sampling of 100 patch applications indicated by our tool yielded 97% accuracy. All these errors were within natural language text (comments or documentation); we are looking into addressing this problem.

Recall, which measures the false negative rate, is much harder to determine in the absence of a benchmark or a codebase where the true patch acceptance information is known. In lieu of either of the above we use the results of analysis of the Apache source code repository by Alonso *et al.* [1]. In CVS log messages on Apache, developers (who commit the change) acknowledge submitters (who provide the change). Of these, we consider only the submissions that came from non-developers; in Apache, these change submissions can be expected to appear on mailling list[2]. This is not a perfect benchmark, since change (or patch) submissions may be received by private email; even if posted on the mailing list, they may be informal change submissions rather than actual patches; or they may be heavily modified before application. However, we *can* place a lower bound on the recall performance of our approach by determining which of the commits containing submissions made by non-developers in the CVS logs were also flagged as files that had successful patch applications by our tool. When evaluating recall in this way, we show a minimum 46% recall rate. Indeed, we conceptually categorize patch submissions into three categories: rejected, accepted, and accepted with modification, the latter of which, in general, is difficult (if not impossible) to recognize. However, for the *accepted* category, we believe that our implementation will give very good results; for the modified category, we'll find those that only have whitespace changes and consistent identifier renamings. It's also possible that some patches were accepted but the proportion of accepted hunks was below our threshold of 3/4. We plan to study the effect of threshold on recall.

Finally, even as we attempt to improve the recall performance (by trying deal with patches accepted after modification) we argue that patches that are accepted without modification are *per se* a phenomena worthy of study, since it indicates a high level of expertise by the patch submitter.

5 Uses of Patch Data

The data produced from datamining patches can shed light on how and why OSS project work. In a companion submission to MSR 2007[3], we mounted a quantitative statistical analysis of how and when OSS mailing list participants became full fledged developers for the four projects studied in this paper. Data such as the number of patches submitted and the percentage of patches accepted was included in the model used. We found that this data represented highly significant predictors in the Apache and Python projects and also improved the model for the PostgreSQL project.

We plan to use this data to help in our investigation of questions such as: Are certain areas of the codebase more prone to patch submission/acceptance than others? This could indicate code with more bugs or code that is more easily understood for newcomers. Are there particular developers who accept most of the patches submitted by non-developers? This may help us understand the roles played by different developers within a project. What is the distribution of patch submissions/acceptances across time and relative to releases? This could aid in examining the development cycle within an OSS project.

6 Conclusion

We have presented a method of collecting data related to OSS communities that to our knowledge has largely been overlooked until now. This data has implications in it's ability to augment analysis of both the software artifact itself and the social community that exists surrounding an OSS project. Although this form of data is just one tile in the mosaic of information that has been gathered with respect to the open source movement, we believe that it is an important one. In this paper we have discussed some of the technical hurdles that have been overcome in the collection of patch related data, given a somewhat preliminary evaluation of our methods, and shared possible uses of this data. In the future, we plan to refine our tools in conjunction with seeking better ways to evaluate them.

References

[1] O. Alonso, P. Devanbu, and M. Gertz. Extraction of contributor information from software repositories. Unpublished `http://wwwcsif.cs.ucdavis.edu/~bird/papers/alonsomsr2006.pdf`.

[2] N. Ducheneaut. Socialization in an Open Source Software Community: A Socio-Technical Analysis. *Computer Supported Cooperative Work (CSCW)*, 14(4):323–368, 2005.

[3] F. W. Lancaster. *Information Retrieval Systems: Characteristics, Testing, and Evaluation.* Wiley, 2nd edition, 1979.

[4] G. von Krogh, S. Spaeth, and K. Lakhani. Community, joining, and specialization in open source software innovation: a case study. *Research Policy*, 32(7):1217–1241, 2003.

[2] Developers may communicate changes privately

[3] Please see `http://wwwcsif.cs.ucdavis.edu/~bird/papers/bird2007obi.pdf`