IEICE TRANS. FUNDAMENTALS, VOL.E95–A, NO.9 SEPTEMBER 2012

PAPER Special Section on Software Reliability Engineering

An Algorithm for Gradual Patch Acceptance Detection in Open Source Software Repository Mining

Passakorn PHANNACHITTA[†a)], ***Nonmember***, **Akinori IHARA**[†b)], ***Member***,
Pijak JIRAPIWONG[††c)], ***Nonmember***, **Masao OHIRA**[†d)], ***and*** **Ken-ichi MATSUMOTO**[†e)], ***Members***

SUMMARY Nowadays, software development societies have given more precedence to Open Source Software (OSS). There is much research aimed at understanding the OSS society to sustain the OSS product. To lead an OSS project to a successful conclusion, researchers study how developers change source codes called patches in project repositories. In existing studies, we found an argument in the conventional patch acceptance detection procedure. It was so simplified that it omitted important cases from the analysis, and would lead researchers to wrong conclusions. In this research, we propose an algorithm to overcome the problem. To prove out our algorithm, we constructed a framework and conducted two case studies. As a result, we came to a new and interesting understanding of patch activities.

key words: *Open Source Software, OSS repository mining, patch acceptance, patch submission, OSS evolution pattern*

1. Introduction

Open Source Software (OSS) has become critical throughout all software development societies. Software developers even from industries take more interest in OSS, because it helps them shorten their software development cycles. More research has been aimed at supporting OSS, which has elevated the variety of studies. Since OSS is a community-driven development, comprehensive-ameliorated studies have become highly popular. There are supportive activities in OSS that enable everyone to collaborate their work efficiently, even they are geographically separated. Patch Submission is the key activity for the purpose. Although an OSS project's archives are manifestly open to everyone, only committers (i.e. core developers who can take the full control of a project) can make a permanent modification to the artifacts. A patch acts like a bridge that connects a gap between committers and non-committers. It enables non-committers to suggest ideas for modifying some parts of an OSS project.

A precise study of patch-related activities was introduced by Bird et al. [1]. They are the pioneers who proposed a method to collect and extract patches for further analysis. Later, there have been many studies following the Bird et al. method that analyzed OSS patch-related information to achieve a better understanding of OSS projects [2]–[5]. Weißgerber et al. [6] have concluded that small size patches are more valuable and suggested the peer developers to submit small size patches for more possibilities of acceptance. However, it has been found that their analyzing procedure was not powerful enough to coverage sufficient cases. They count a patch as accepted if the whole of its record was committed into a revision at once. This led them to their mistaken conclusion that smaller size patches are preferable.

This study is motivated by the studies of Bird et al. [1] and Weißgerber et al. [6]. We have found that their analytical grain may be too coarse to conclude the characteristic of acceptable patches (i.e. the preferred size of patches for acceptance). The question is what will happen if patch activities analysis is performed in finer grain? In this study, we devise a novel algorithm that divides patches into smallest meaningful grain, which is a line of source code (LOC). It led us to discover new dimensions of patch analysis. In our evaluation, we certify the preferable size of accepted patches. Our devised algorithm enables us to include the case in which only a portion of the source code was accepted gradually, which is more likely to be occurrence in practice. With further evaluation we try to explore a totally new finding that can be brought out only from our propose. We seek out an inconclusive conjecture that fine-grained patch acceptance analysis has a potential to elucidate it. We turn out that temporal-based patch submission analysis could explain OSS project evolution and can conclude an inconclusive belief from Nakakoji et al. proposed OSS evolution pattern [7].

The remainder of the paper is organized as follows: Sect. 2 outlines background related to the analysis of patches submitting activities. Section 3 explains our proposed algorithm and its method of implementation. Section 4 demonstrates our case studies and elaborates the results. Section 5 discusses on our findings and validates our proposal, and Sect. 6 concludes the paper.

Manuscript received December 25, 2011.
Manuscript revised April 20, 2012.

[†]The authors are with the Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma-shi, 630-0192 Japan.
[††]The author is with the Faculty of Computer Engineering, Kasetsart University, Bangkok, Thailand.
a) E-mail: phannachitta-p@is.naist.jp
b) E-mail: akinori-i@is.naist.jp
c) E-mail: b5005135@ku.ac.th
d) E-mail: masao@is.naist.jp
e) E-mail: matumoto@is.naist.jp
DOI: 10.1587/transfun.E95.A.1478

2. Background

2.1 Project Repository

A software repository is a system that enables a truly community-led development. According to the principle of OSS, every artifact inside the system should be manifestly open to the public. The project repository holds every artifact from the beginning of the project imprinted with each file's revision. This imprinted revision allows everyone to start a development from any snapshot in any prior revision. Furthermore, the difference between any two previous revisions enables everyone to track an improvement of a project component from any artifact.

Recently, software repository management tools have been improved notably. To date, there are many well-known software repository management tools such as CVS (Concurrent Versions System), SVN (Subversion), and git. They also provide many useful features. Their usability becomes highly effective and efficient, encouraging developers to work on an OSS project with ease. Likewise, those such provided features also give researchers an opportunity to study and analyze the project artifact thoroughly for a better comprehension on the OSS projects and societies.

2.2 OSS Patch

Patches are an important in filling gaps in OSS development societies. Without patches, it would be too complicated for anyone to understand the requirements or ideas of others. Patches have become a universal standard protocol for all the communications about of changes.

When a non-committer needs to suggest a code-base idea, they just modify the source codes. They then create their patch from their changed source code, and submit it to the project committers. A patch is usually sent through conventional channels such as a bug tracking system (i.e. Bugzilla) and a mailing list. The submitted patch will be discussed on those channels, allowing all to brainstorm on that submitted patch together. If the discussion reaches a consensus, the patch will be applied to the repository and the modified original file will be released with an upcoming version. Whether a whole patch or just a portion of it has been submitted, we refer to this as patch acceptance. For easier understanding of our proposed fine-grained algorithm, we first explain patch creation.

2.2.1 Patch Creation

OSS Patches contain different LOC in the original version and the developer's, so we sometimes call the un-submitted patches diff files. The different LOC is indicated in patches line by line, showing what is added, deleted, or changed. To date, there are 3 distinctive formats of diff files implementation. They are Standard diff, Unified diff and Context

1	def greet(name)	1	def greet(name):
2	print 'hello', name	2	print 'hello', name
3	greet('Alice')	3	def getname():
4	greet('Bob')	4	return raw_input()
		5	greet(getname())
//repos/m/hi.py (revision 1.4) Last Modified: Sep, 12 2011 11:11:01		//local/m/hi.py (modified in local) Last Modified: Sep 15, 2011 23:56:08	

Fig. 1 An example comparing changed source code and its original file.

diff formats. Each type of format comes from the development of repository management tools. Although Standard diff is the simplest and can be utilized in practice, it consists of too little information for any further proposal [6]. Moreover it is impossible for a researcher to perform any kind of study and analysis using Standard diff information, if it was sent as a patch. Fortunately, that lack of information has brought widespread OSS development. Few Standard diff have been found in the bug tracking systems recently, so that researchers can omit them from their study [6].

Figure 1 shows an example of a modified python file. The left side is the original file, /m/hi.py, that was downloaded from the software repository, and the right side is modified by a developer. There are 2 LOC deleted and 3 added. Figure 2 shows three types of diff files that are created from both files seen in Fig. 1. As shown in Fig. 2, Standard diff at least has not provided the modified timestamp, which is necessary for any study of patch-related activities.

2.2.2 Patch Acceptance

If any portion of a patch has been committed to the project repository, it means the committers have accepted a suggestion from outside. In fact, committers can decide to accept only some portions, and they can also gradually commit a patch in several revisions. This always happens when a submitted patch is large, when it is likely to consist of many components. For example, if a patch contains 100 LOC with 10 methods and is committed 10 LOC each in 5 revision, it is 50% accepted by the committers. However, there is no such algorithm in all of the existing studies that can include a gradual patch acceptance case in an analysis.

3. Algorithm for Gradual Patch Acceptance Identification

This section elaborates our algorithm on how to include gradual patch acceptance for further patch-related study. We also explain thoroughly a framework construction derived from this algorithm. First of all, Fig. 3 gives an overview detail of our proposed algorithm in scenario.

Suppose that a non-committer 'A' checked out foo.py from the project repository. Now the latest committed foo.py is imprinted with a revision num 42. 'A' then removes a LOC contained "bbb", and adds two LOC containing "yyy", and "zzz". After 'A' creates a diff file, foo.diff, from his foo.py and the original foo.py, his foo.diff consists of 3 records of changed LOC. He needs his change to be

Format	Standard Diff	Unified Diff	Context Diff
Command (CVS)	cvs diff -r 1.4 /m/hi.py	cvs diff -u -r 1.4 /m/hi.py	cvs diff -c -r 1.4 /m/hi.py
Header (CVS)	Index: hi.py ========== RCS file: /m/hi.py,v retrieving revision 1.4 retrieving revision		
Modified time		- - - hi.py 12 Sep 2011 11:11:00 -0000 1.4 +++ hi.py 15 Sep 2011 23:56:08 -0000	

Modified contents	3,4c3,5
	< greet('Alice')
	< greet('Bob')
	—
	> def getname():
	> return raw_input()
	> greet(getname())
		@@ −1,4 +1,5 @@
		def greet(name):
		  print 'hello', name
		-greet('Alice')
		-greet('Bob')
		+def getname():
		+ return raw_input()
		+greet(getname())
			**********
			*** 1,4 ****
			def greet(name):
			  print 'hello', name
			! greet('Alice')
			! greet('Bob')
			— 1,5 —-
			def greet(name):
			  print 'hello', name
			! def getname():
			! return raw_input()
			! greet(getname())

Fig. 2 Information obtained from each kind of the diff files.

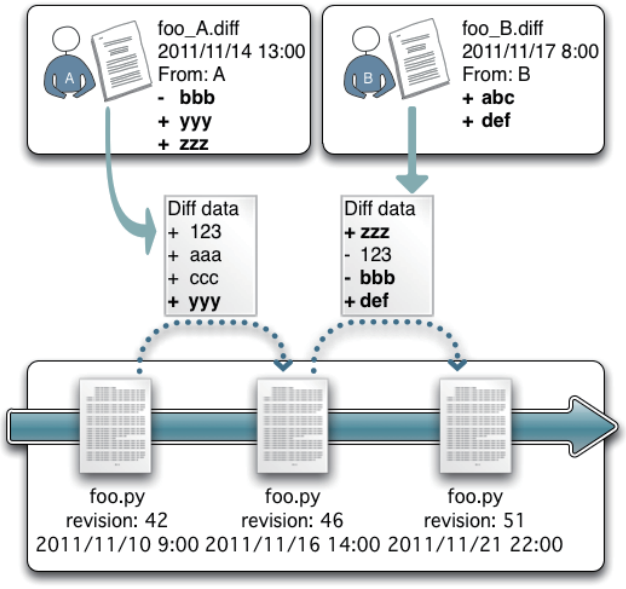


Fig. 3 Overview scenario.

applied by everyone who will use this source code in the future, so he submits foo_A.diff as a patch into the project bug tracking system. Suppose that he submits foo_A.diff on Nov 14, 2011at 1:00 pm. Later, a LOC in the foo_A.diff patch has been committed in revision 46, and another two LOC are committed in revision 51. At a snapshot on revision 46, this foo_A.diff has been accepted 33%, however; 5 days later it has been fully accepted in revision 51. Another non-committer 'B' also checked out foo.py and made 2 changed LOC by adding "abc" and "def" in foo.py. He also made his foo_B.diff and submitted it through the project bug tracking system on Nov 17, 2011 8:00 am. At the current revision (revision 51) only "def" and been committed, so that this foo_b.diff is concluded as 50% accepted.

How can an algorithm detect the above scenario? Suppose that we investigate every different LOC between each contiguous pair of revision snapshots after the submitted timestamp of patches, the committed records will be turned out. Thereafter, if we compared each committed record with the submitted patch and count the identical, both portion acceptance and gradual acceptance can be detected and included into any kind of further analysis.

To implement and evaluate our proposed algorithm, we construct a viable framework. The framework is divided into 3 phases. The first two phases are data extraction. We name them Patch extraction and Repository's diff-file creation. The last phase is our proposed algorithm performance, which we name Gradual patch acceptance identification.

3.1 Patch Extraction

To cover both conventional patch submission channels, Bugzilla and mailing list, patches from both sources need to be gathered correctly. The two types of sources contained patches in totally different structures. So we need two particular patch extraction methods. There are several existing studies of patch extraction from a mailing list [8]–[10]. We choose to improvise and extend the patch extraction phase proposed in Weißgerber et al. [6]. Their method is straightforward and achieved sufficiently high precision. Patch gathering and extraction in Bugzilla, on the other hand, has no obvious proposal. Moreover, the patch attachment locations in Bugzilla are highly diversified, so that a

sophisticated method is needed for this purpose. We developed a focused website crawler augmented with heuristic rules to cover all possible attachment locations.

3.1.1 Patch Extraction from Mailing List

In a mailing list, patches are always mixed with normal text in the mail body. We can set them apart by following the Weißgerber et al. proposal [6]. The goal of this phase is to transform raw patch data into a good format for a further analysis. As ing existing studies, we exclude all the Standard diff format that has insufficient information. We break Context diff and Unified diff into readable LOC grain, and collect all necessary information that related to a LOC and all the patches.

Let a tuple $(I_p, P_p, t_p, L_p, [c_p])$ denote $patch_i$, where I_p is patch's index, P_p is patch's absolute path, t_p is patch's submitting time, L_p is the total number of the changed LOC, and $[c_p]$ is a list containing all the individual changed LOC in $patch_i$. In more detail: I_p is a short label that describes a patch. P_p is what we can identify as a corresponding original file in the repository. The timestamp of each patch can be indicated with t_p. There is a common issue in temporal-related analysis when analyzing data sources come from different time zones. Timestamp needs to be converted into a common timezone such as UTC. Otherwise an analysis will be inaccurate. In this work we have found many types of time zones such as UTC, EST and EDT, and we then parsed and converted all of them into UTC. Next, we can use L_p to label and categorize the size of patches. Finally, each member in $[c_p]$ is our further analysis grain, which is the smallest meaningful grain of patch-related activity analysis.

3.1.2 Patch Extraction from Bug Tracking System

Bug tracking system is mostly implemented in web applications such as the well-known Bugzilla. A web crawler is needed to gather patches from it, and, moreover, a web parser is required for the content extraction. Since all the file types are allowed to submit into the bug tracking system as patches, it is better to develop a focused web site crawler [11] for this purpose. We augment a heuristic to our focused bug tracking system crawler that works as follows. In each crawled bug report, firstly, the crawler parses all html tags off using an Html-parsing tool named Jericho HTML parser, which is a magnificent HTML parser implemented in Java [12]. Next the crawler rolls out the attachment section in that page, and all the text-base and archived files are collected. We recursively extract each archive file to collect all the text-base files in it. After we collect all the attached text-base files of a bug, we roughly filter out the non-patch file by looking up each file thoroughly for a patch indicator keyword. Keywords are "Index:", "RCS file:", and "diff". After we filter out the possibly non-patch file once, we perform a finer filtering by looking for an indication of the original file's absolute path corresponding to that patch. If the absolute path is found and the original file in repository can be located by that path, we conclude that the attached file is a submitted patch. After this step, the patch extraction from a text-base file can be adapted from the mailing list patch extraction method. After all patches are extracted, we compose them into records and store them into a database.

##Rev.No.	Committed time-stamp	
51	2011-11-21	22:00:59
46	2011-11-16	14:00:34
42	2011-11-10	09:00:12
19	2011-10-14	21:00:01

Fig. 4 Committed revision number and time-stamp pair example.

3.2 Repository's Diff-File Creation

These diff-files are created manually, with each contiguous pair of revision snapshots of every file contained in the project repository. At the beginning, all the OSS project source code files are checked out from the repository. Next, we create a list of tuples consisting of committed revision number and committed time-stamp pair, as in the example in Fig. 4. We then create diff files between each contiguous pair of revision records.

Created diff file is also treated as a tuple similar to an extracted patches. A tuple $(I_r, P_r, t_r, [c_r])$ denotes a Diff file record. I_r is a created diff file's index, P_r is a source code's absolute path, t_r is its timestamp. We also parse the timezone into a common UTC. $[c_r]$ is the list of changed LOC between a pair of a revision. We also compose $(I_r, P_r, t_r, [c_r])$ into records and store them in a database.

3.3 Gradual Patch Acceptance Identification

To note if a patch is gradually accepted, we need to find out if a LOC record in $[c_p]$ can be found in $[c_r]$ any time after t_p. First of all, we need to specify which original file in the project repository is $patch_i$ is submitted for. In fact, the index field in both patches and diff files are designed for this purpose, so we only need a comparison between $I_p = I_r$ to solve the specifying problem. However, the index field has often been omitted by the patch submitters, which requires us to match between the absolute path (P_r and P_p) instead. Unfortunately, the absolute paths from both sources are different, because patches are made in different environments. The best we can do is to perform the longest string matching between P_r and P_p. In the case that matching result turns out more than one candidates, we omit that LOC because of an ambiguity.

To make the matching more reasonable, we define a time scope variable named Δt, In fact, there must be a number of submitted patches per day (the screening process can take some time), so as to minimize the time required for a patch to become accepted. We define Δt from our assumption that the screening process has not longer than Δt days, which makes our proposed algorithm more capable from noises. (That is, there may be a duplicate LOC that

is committed a couple years after $patch_i$ has been submitted. In this case we could misjudge it as a record of $patch_i$'s acceptance.)

We conclude that a $patch_i$ is an accepted patch if there is an existing LOC that has been committed to the repository within a defined Δt scope.

$$\begin{array}{ll} & (I_p \quad = I_r \vee P_p \sim_{match} \quad P_r) \\ \wedge & \quad t_p \quad + \Delta t \leq t_r \\ \wedge & (\exists l \ \ |l \in [c_p]) \subseteq [c_r] \end{array} \tag{1}$$

Note that each LOC in both $[c_p]$ and $[c_r]$ has been whitespace-collapsed. We have a strong presumption that in gradual patch acceptance, committers often apply patches manually. They can produce some whitespace shifts that can produce an accuracy loss in the identification process.

4. Case Studies

We have conducted two case studies to show potential utilization of our proposed algorithm. The first was to study patch acceptance characteristics when including the gradual patch acceptance case. The second case study analyzes the temporal-based patch commitment in its gradual patch acceptance aspect.

4.1 OSS Project Data Sources

Both case studies are conducted on two well-known OSS projects, the Apache HTTP Server and the Eclipse Platform. Apache HTTP Server is an instance of data source for a project that utilizes SVN as source control and mailing list as patch's communication channel. On the other hand, Eclipse Platform, is an instance for CVS repository and Bugzilla. After the data extraction in the first and second phases has been finished, there are two databases containing records as shown in Table 1. The Eclipse Platform dataset is approximately eight times larger than the Apache HTTP Server dataset, so we denote Eclipse Platform dataset as a large dataset and Apache HTTP Server as a small dataset.

4.2 Patch Acceptance Analysis

We chose an acceptance rate for each $patch_i$ as our measurement. Acceptance rate can be calculated from the extension of gradual acceptance identification, by calculating the ratio between the summarization of each LOC which is held by Eq. (1) in a patch and its L_p as illustrated in Eq. (2).

$$AcceptRate(patch_i) = \frac{\sum l \mid (l \in [c_p] \longrightarrow l \in [c_r])}{L_p} \tag{2}$$

To make an observation more obvious, patches should be categorized into several classes by size. We decided on 5 classes of patches that rather grow in exponential. Their ranges are as follows, 1-4 LOC, 5-19 LOC, 20-49 LOC, 50-199 LOC, and 200 LOC and higher. We denote (0,5) LOC as small patches to be the same as in the existing study [6]. Also, we define the level of patch acceptance in classes of percentage. We categorized percentage into two main classes: partial and fully acceptance. Partial acceptance is divided into quartiles that are the four ranges of acceptance:(0%, 25%), [25%, 50%), [50%, 75%), and [75%, 100%), and full acceptance is only 100% acceptance. For example in the class of 20-49 LOC, the [50%, 75%) quartile indicates that patches are accepted between 10 and 36 LOC. Note that, all of the ranges include the gradually accepted cases.

Table 1 Transformed data from both projects.

(a) Repositories

	Apache HTTP Server	Eclipse Platform
Repository	SVN	CVS
Observing period	1998/01 - 2003/12	2002/1 - 2007/12
#File	6,685	47,968
#Changed line	2,223,487	11,012,316

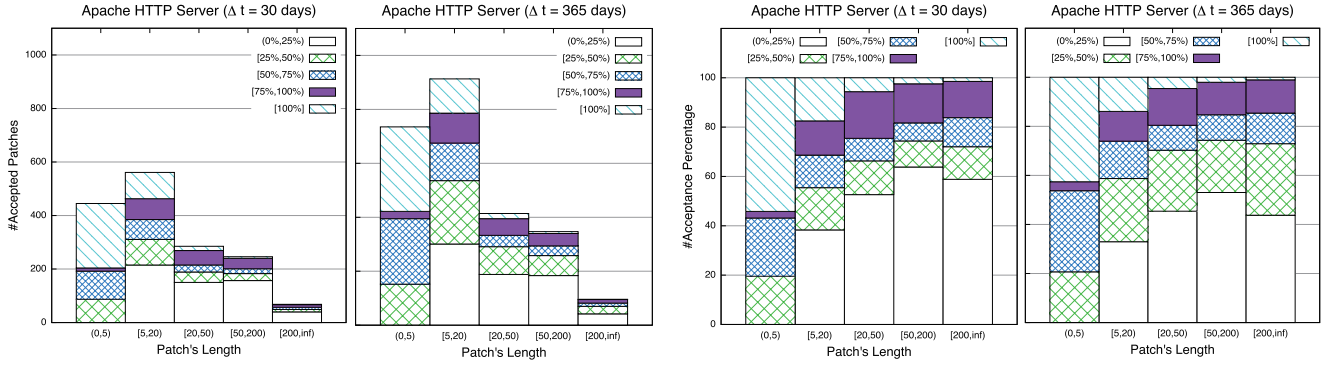
(b) Patches Data Source

	Apache HTTP Server	Eclipse Platform
Source	Mailing list	Bugzilla
Observing period	1998/01 - 2003/12	2002/1 - 2007/12
#Patch	6,370	89,513
#Indicated Original file	2,240	84,949
#Changed line	171,354	13,086,116

4.2.1 Experimental Results

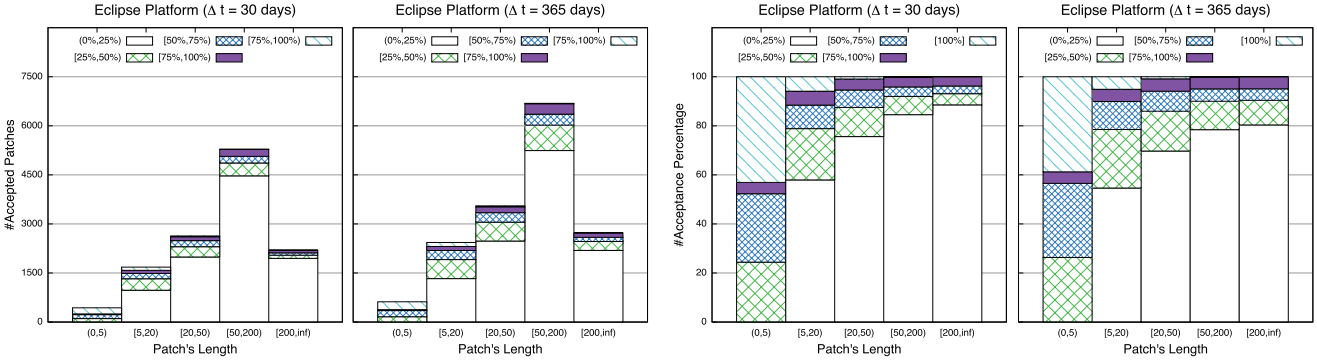
We set up a variation of Δt as 30 and 365 days; hence both time scopes are meaningful (i.e. a month, and a year). We start an experiment on our small dataset setting up with both Δt. The results from the gradual patch acceptance identification are shown in Fig. 5. In Fig. 5(a), the x-axis indicates 5 classes of a patch's length classified by L_p (i.e. The leftmost stacked bar signifies the small patches acceptance.). The y-axis indicates the total number of accepted patches. For each stacked bar, there are 5 classes of patches acceptance percentages. The bottom area indicates a low value of acceptance (i.e. under 25% accepted). The top area indicates the full acceptance. The obvious results pointed out in this experiment is the both set up Δt as 30 days and 365 days yields the same result in patch acceptance point of view. For instance, our proposed algorithm can show an improvement to achieve more insights of patch acceptance observation. The insights are (I) the numbers of full acceptances are decreased by the incremental of size of patches. Also (II) the full-acceptance class as shown in the top area seems to be less than the others. In this experiment, there are more accepted patches among the small patches class when compared to the larger patches in the Apache HTTP Server project.

To be more concrete, the stacked histograms of percentage plotting are more distinguishable. The percentage plotting of the Apache HTTP Server experimental result is illustrated in Fig. 5(b). In this figure, the x-axis indicates 5 classes of the patchs length classified by L_p the same as in Fig. 5(a). The y-axis indicates the acceptance percentage for each class of patches from 0% to 100%. Figure 5(b) assures the minority of full-acceptance class. Some concrete



(a) Counting the number of accepted patch from Apache HTTP Server dataset (b) Acceptance percentage from Apache HTTP Server dataset

Fig. 5 Experimental results from Apache HTTP Server dataset.



(a) Counting the number of accepted patch from Eclipse Platform dataset (b) Acceptance percentage from Eclipse Platform dataset

Fig. 6 Experimental results from Eclipse Platform dataset.

observable points in Fig. 5(b) and that few of the patches are accepted most of their LOC (i.e. If small patches are excluded as a outliner, less than 30% of accepted patches are accepted over 75% of their LOC.)

We performed another experiment on the large dataset, Eclipse Platform, since the characteristics of the two projects are very different. For example, the total submitted and committed LOC during 5 years of observation is very distinctive, as is shown in Table 1. Figure 6 shows the result of counting the number of accepted patches. The trend of accepted LOC in the Eclipse Platform and the Apache HTTP Server is different as seen in Figs. 5(a) and 6(a). Larger patches are accepted more in Eclipse Platform projects. However, it is very interesting that in the percentage plotting shown in Fig. 6(b), trends in both projects are identical, as is shown in Figs. 5(b) and 6(b). The full-acceptance class is also a minority among others, as they are in the Apache HTTP Server.

4.3 A Study of Temporal-Based Patch Commitment Analysis

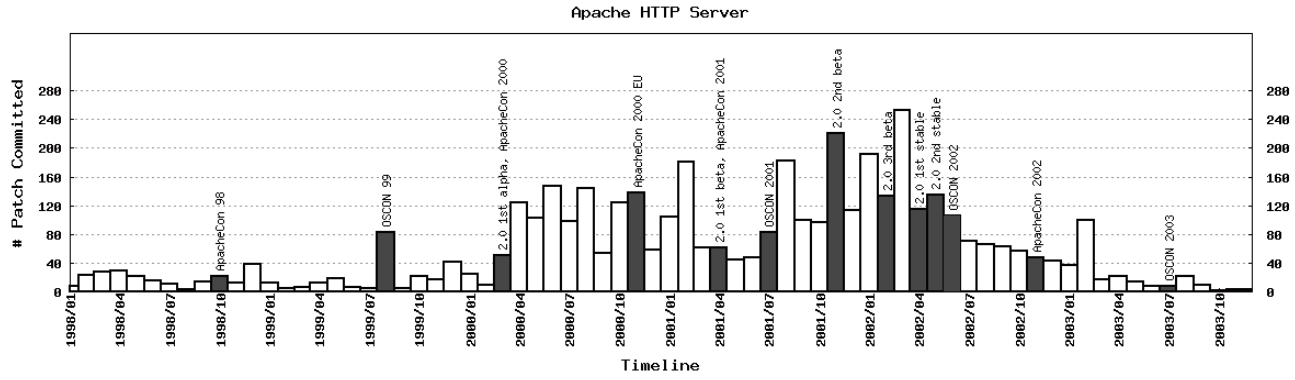
In this case study, we study the change of patch commitment rate over time. First, we collect the accepted patch commitment record, and we count the total number of patches which are committed in each month. The counting-in condition is shown in Eq. (3). In this study, the Δt variable from Eq. (1) is set as 365 days.

$$\begin{aligned} \#CommittedPatch(month_i) = \sum Patch_i \mid (\exists l \quad \mid \\ l \in [c_p] \rightarrow l \in [c_r] \\ \wedge \quad month(t_r) = month_i) \end{aligned} \tag{3}$$

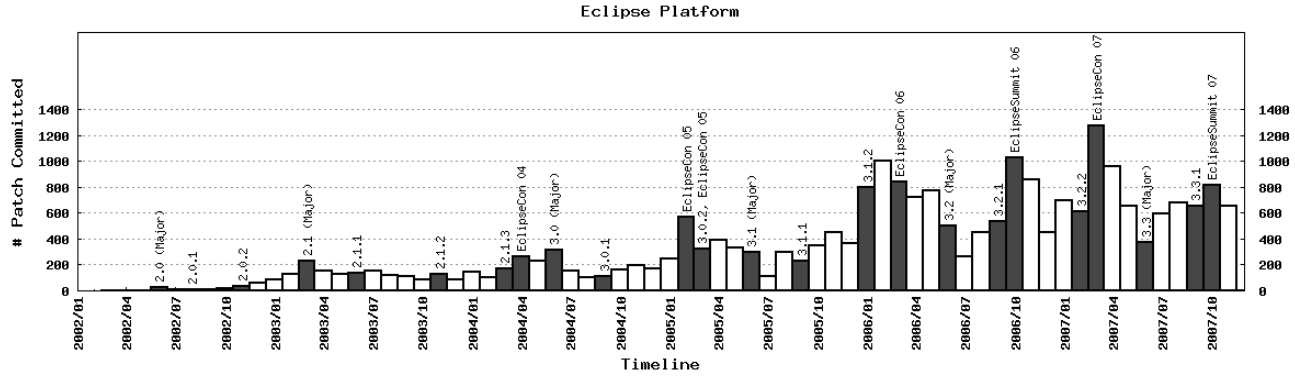
Figure 7 shows preliminary results from the Apache HTTP Server and the Eclipse Platform dataset. Preliminary results from both datasets are shown in Fig. 7. Note that, the highlighted bar and the corresponding labels above it will be used for further discussion. A notable diversity of patch commitment level in each month is very interesting. The committers must have been influenced by some factors to vary their openness level [13]. (Openness refers to how much committers accept, share and comply with non-committers.) In some months, committers are suddenly more open, which makes us focus on the temporal factor. From the characteristics of Fig. 7, we suggest that special event occurrence and the changing trend have a possibility to be the factors.

4.3.1 Analytical Result

In Fig. 7, we can notice two types of alternation. The first



(a) Analyzing patch acceptance from Apache HTTP Server project from January 1998 until December 2003



(b) Analyzing patch acceptance from Eclipse Platform project from January 2002 until December 2007

Fig. 7 The accepted patch commitment in Apache HTTP Server and Eclipse Platform project.

one is pulses (i.e. Apache HTTP Server in October 1998 and August 1999 in Fig. 7(a)). And the second one is the number of commitment changes after a pulse.

Since pulses are a transitory alternation of openness level that seems to have a high impact. It makes us think of a special event. We started an investigation on both projects. We found the record of conferences held and versions released. They can be matched with the two types of pulses in Fig. 7. The investigation on pulses, which do not have a long effect on changing the openness level (i.e. the number of committed patches), turned out many records of conferences. Matching the record into Fig. 7, conference makes the committers openness level increased in a month or the next month, and then the openness level returns into the same level as before the conference. For example, Apache HTTP Server in October 1998 and August 1999 in Fig. 7(a) are pulses, which keep the openness level in the months before and after at the same level. It is very reasonable that the OSS community of a project must have placed some advertisements for the forthcoming conference (i.e. calling for papers). It would arouse the newcomers as well as incite the activeness of the currently inactive developers to participate with the OSS community.

On the other hand, a pulse in the month in which a released version was held (i.e. Eclipse Platform in June 2004 and March 2005 in Fig. 7(b)) changed the openness level afterward, and also has a long-lasting effect. In Fig. 7, we highlight a bar of the months in which an OSS community held special events and label the event name above. There is a difference in the increasing and the decreasing number of patches commitments. When a project has released a stable version, it must have been stable enough for fewer issues to be still opened. So, the committed level should be continually decreased afterward. In contrast, a released of an unstable version (i.e. alpha or beta version) should still have a lot of remaining issues that need much discussion and refinement. So, the openness level after a release of an unstable version is continually increased.

Note that we have excluded all the records of Apache HTTP Server 1.0 released versions from our investigation, because a new version was released almost every month the first couple of years.

We have figured out that the changing of openness level can explain an OSS project's state of development, so we try to combine it with another temporal factor: trend. We found one study of OSS evolution very interesting [14]–[16], since it is not only a temporal-based study, but is also related to our question. From the bibliographical surveys, an OSS evolution pattern proposed by Nakakoji et al. [7] is the most fascinating and suitable for deriving from our possessed information. Nakakoji et al. classified OSS projects into 3 types and suggested a transition between each type. The three types of OSS projects are Exploration Oriented, Utility Oriented, and Service Oriented. Table 2 shows a sum-

Table 2 Three types of OSS project.

	Exploration Oriented	Utility Oriented	Service Oriented
Objective	Sharing Innovations and Knowledge	Satisfying an individual need	Providing stable service
Control style	Cathedral-like central control	Bazaar-like decentralized control	Council-like central control
Community structure	Project leader and core members	Core members and many peripheral developers	Core members and Many passive users
Major problems	Finding a novel innovation	Difficult to choose the right program	Less innovation

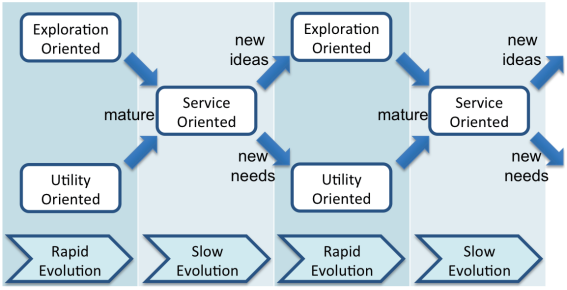


Fig. 8 Nakakoji et al. proposed OSS Project's Evolution pattern [7].

mary of their properties. They also believed that the evolution between each type will be alternated as shown with the transition in Fig. 8, and their belief still needs supportive evidence.

There are three existing trends in the commitment record in Fig. 8. They are increased, decreased, and unchanged openness levels. We map between the trends with an evolution pattern using the properties in Table 2. Bazaar-like decentralized control is the only type that can make a notable increase in the number of committed records [17]. We presume the project must be a Utility-Oriented during that period. We can also conclude that the special event pulses period is transitory Utility-Oriented, because the number of committed records also increased significantly. Next, the decrease in committed records usually occurred after a major release. We presume that the project is going to become stable, so that there are fewer remaining defected that needed a fix. Consequently, the objective of the Service-Oriented as shown in Table 2 makes it be a perfect conclusion for this case.

From this observation, there are three cases that can elaborate the unchanged openness level. The first case is when a project has just been founded or is well-known. Because the project is not very active, this case can be simply considered as an Exploration-Oriented. The second case is an unchanged openness level which is found after a Utility-Oriented period. We presume that most of the satisfied requirements have already been fulfilled that make the project more mature and less active. We conclude this case is a transition between Utility-Oriented and Service-Oriented.(Indicated as mature in Fig. 8) The last case is when it occurred after a Service-Oriented period. After a project has been mature for a little while, a new need of expansion would become a topic of development. It can be in a transition state becoming an Exploration-Oriented or a Utility-Oriented. It will be a transition to an Exploration-Oriented if new ideas are the most critical. On the other hand, if there is an urgent need for many new features, it would probably be in transition to becoming a Utility-Oriented.

5. Discussion

5.1 Gradual Patch Acceptance Identification Algorithm

In fact, the existing development of all patch acceptance detection methods are in reversed engineering. The more criteria in practical were included, the more we can comprehend with the activity and the OSS society as well. Our proposed algorithm is devised from a better hypothesis that give a better reflection to actual. Our two different extensions based on the algorithm have already shown that after the overlooked criterion was included into the identification, many consequence are turned out difference. Moreover, an unexplored area of study can also be discovered.

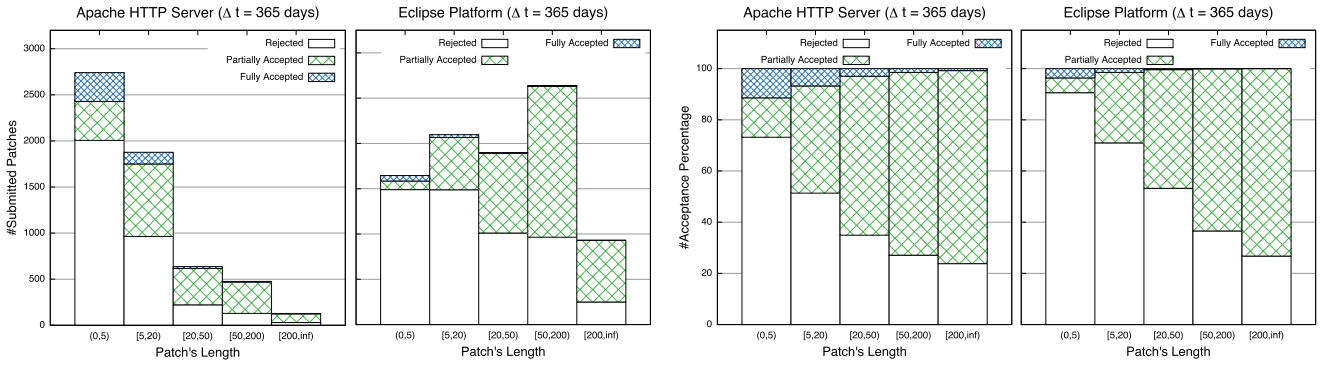
There are still more criteria that need to be studied, for example the variable name substitution. The methods in the existing studies have already tried to indicate a substitution of the variable name [1], however non of them are good enough for the practically use. Because the provided log itself does not contain any helpful information to find them out. All the approaches are using too simple heuristic that also unable to evaluate it clearly. Moreover, it is impossible to know that a patch was accepted if committers refine the whole submitted patch before they commit it. For that case, our devised algorithm at least can detect the untouched LOC that can give out an acceptance record instead of rejection. However, we believe that such case is rather to be omit than the gradual patch acceptance case, which is likely to be happened more frequently.

5.2 A Discussion on Patch Acceptance Analysis

The following are discussing what we have discovered along with the patch acceptance analysis that extended from our devised algorithm.

5.2.1 Larger Patches Can Be Concluded as More Accepted if Rejected Rate is Considered

Rejected rate is also an interesting feature in patch acceptance analysis. It has never been discussed in any existing work. Since the conducted experiments discussed only acceptance rate are still unsatisfactory to tangibly conclude if larger patches are more preferable than smaller patches, we perform one more experiment included rejected rate. In this experiment, we categorized patch acceptance into 3 classes. They are fully accepted (i.e. 100% of LOC accepted), partial accepted, and rejected. We perform this additional experiment using only 365 days of Δt. Figure 9 shows the



(a) Counting the number of submitted patches

(b) Including reject rates into percentage plotting

Fig. 9 Experimental results including reject rates.

result. In Fig. 9(a), the x-axis is indicates 5 classes of patchs length classified by L_p as in Fig. 5(a). The y-axis indicates the total number of submitted patches. The top area in each bar indicates the fully accepted class. The middle area indicates the partial accepted class, and the bottom area indicates the rejected class. From this figure, we can find the difference of the major size of the submitted patches from both projects. Apache HTTP Server project developers tend to submit small patches over large patches. However, in both projects, larger patches seem to be more accepted.

To conclude if large patches are more preferable, percentage plotting is more suitable. Figure 9(b) shows that there are two interesting common characteristics in both projects. First, the number of rejected patches is far more than accepted patches. Second, when we focus on accepted patches area in Fig. 9(b), larger patches can be concluded as more accepted, which is a different conclusion from the existing study proposed by Weißgerber et al. [6].

In conclusion, our algorithm is flexible enough to compare our hypothesis on gradual acceptance with the existing fully acceptance case [1], [6]. In Figs. 5(b), 6(b) and 9(b), if we focus only the top area in each bar, which indicates only the fully acceptance case, we can observe that the acceptance trends from both projects are identical that the fully acceptance rate is decreased by the size of submitted patch. This can certify the correctness of the existing study [1], [6], which the authors have a right conclusion derived from the existing patch acceptance detection method. However the different outcome is turned out, when we include those aforementioned overlook cases. It leads us to conclude that the patch acceptance rate increases by the size of submitted patches in actual.

5.2.2 The Varied Time-Scope (Δt) Certifies the Existence of Gradual Patch Acceptance Case

In both Figs. 5 and 6, the set up Δt variable has effected the patch acceptance rate. The longer Δt derivers more acceptance rate beside the respected identical trend. It is obviously that the gradual commitment of patches has generated it. Note that, we certainly aware the duplicate LOC counting for an accepted patch, so that we believe the acceptance gains from $\Delta t = 30$ to $\Delta t = 365$ is sounded. Because larger patches should consists of more components. The more components are contained in a patch, the more dependencies it needs to be clarified, so it will need more time to screen or wait for an appropriate time to be used. It makes us conclude if a patch is judged as accepted within only one commission is not a sounded conclusion.

5.2.3 An Elaboration on the Categorized Size of Patch

We mention briefly about the size of patch categorization in the above section that it is distributed in exponential. In addition, each range has its inherent meaning. Small patches (less than 5 LOC) would be just a minor modification. We presume that the minor modification would rather be reference than other larger modification, so that it would inherently has more acceptance possibility. The length between 5 and 20 LOC is an average LOC of interface or method modification. Between 20 and 50 LOC starts to be a length of a component modification, which may contain some error correction. From 50 to 200 LOC is likely to indicate the modification of a large component, which may consist of some dependencies. For 200 LOC and more, we believe it is submitted for a severe defect fixing, or re-implement the whole component from the scratch. The component would be a huge one, that would probably consist of many modules. These elaboration shows the more difficultly of larger patches to be accepted the whole and committed at once. The result turned out from the 200 LOC and more class also validate the noise-prone property for our propose framework. Since over 100 LOC must be committed to conclude 200 LOC and more class as over 50% accepted. It also makes us believe the noise-proneness and the accuracy of this case study.

5.3 A Discussion on a Study of Temporal-Based Patch Commitment Analysis

Comparing the records of accepted patch commission between two projects in Fig. 7, Eclipse platform is more predictable. In Fig. 7(b), Eclipse platform is more respected

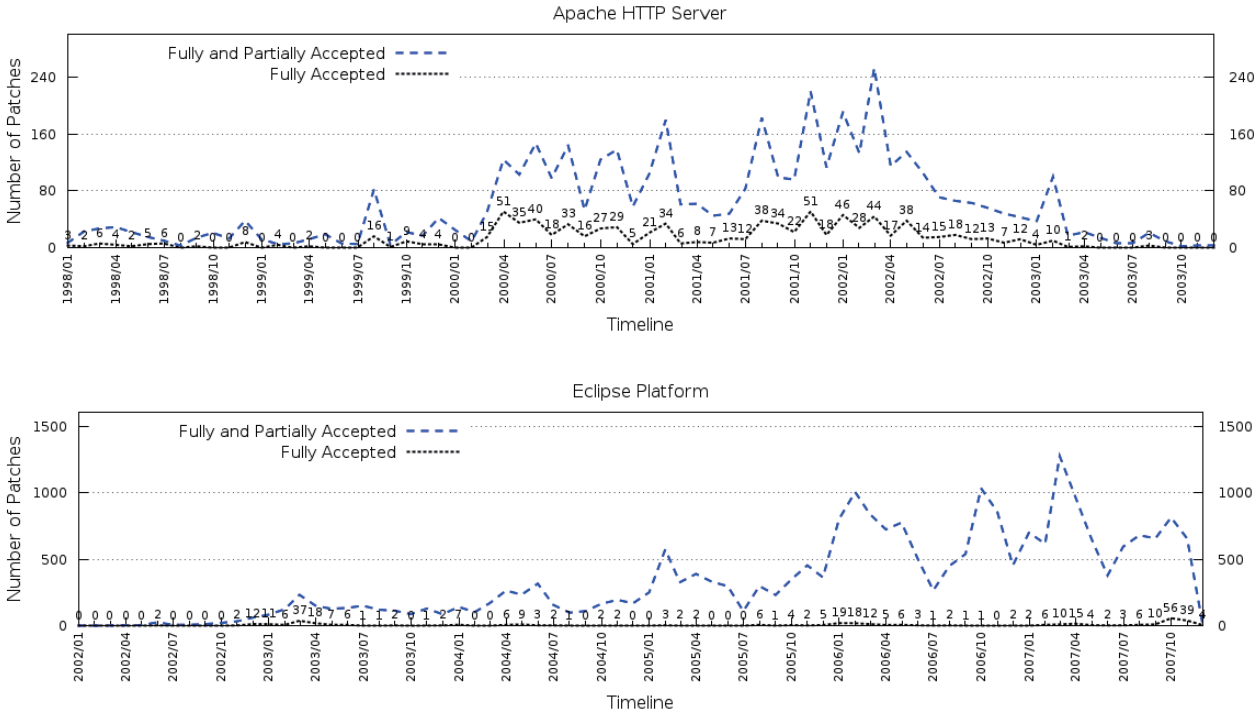


Fig. 10 Comparing submitted and committed patches in Apache HTTP Server and Eclipse Platform.

to our finding conditions. For example, the local highest relative peaks are always existed in the month that held a conference as well as the openness level also decreases after a major released. We believe that Eclipse Platform has a clear declaration of its development cycles, which is in pattern. (i.e. the whole cycle spans into a year, which begins in March) Eclipse Platform always releases a new major revision every year around June. Since Eclipse Platform participant knows the well-declared project's check point, they can work and evaluate their work product more effective and efficient. They will be able to estimate how much they need to exert their effort to bring the project to reach the annual goal. The clear development cycle declaration of an OSS project also make it possible to perform a self-comparable, which is a good approach to appraise a project from cycle to cycle.

There is an additional interesting observation on Eclipse Platform project. In 2006, Eclipse platform introduced one more annual conference (Eclipse Summit) in 2006. Since then the openness cycle have been shorten from one year into half year cycle (i.e. from March to October and from October to March), which is still predictable. In conclusion the a clear declaration of its development cycles will provide more reliability to an OSS project.

5.3.1 Comparing the Result with the Existing Algorithm

To show the value of utilizing our proposed algorithm corresponded to this study, we conduct an experiment with an implementation of the existing coarse-grained algorithm. Figure 10 shows the result. The x-axis of both graphs are the timeline in terms of month, and the y-axis means the total number of committed patches. In Apache HTTP Server graph, we can see an existing of pulses if we consider only the fully acceptance case. Pulses on the months that holding a conference is quite notable, however, the trend of openness level is undistinguishable enough to study the OSS evolution pattern. Eclipse Platform is the better example. Eclipse Platform graph in Fig. 10 shows that it is not enough to study the temporal-base patch commitment with patch submission records. From the acceptance rate result in Fig. 6, Eclipse Platform has a very few number of fully-accepted patches. It makes us conclude that coarse-grained algorithm is not powerful enough to study the temporal-base patch commitment (i.e. OSS evolution pattern achievement).

5.3.2 Analytical Result Verification

Since we use mailing lists as patches source in Apache HTTP Server dataset. In a mail body, it often contains with talks or discussions, that makes us trace whether the developers were talking about events. We randomly sampled mails from the mailing list and read them thoroughly, we have found out that there are many talking related to event that was held in a corresponding month. We can find both before and after an event discussion. For example, many emails were talked about an upcoming conference, furthermore, some people advertised about the upcoming events on their email's signature. To validate the gradual patch acceptance from accepted patch commitment, we observed the total number of patch submission in each month. Figure 11 shows that there is an existing month that the number of

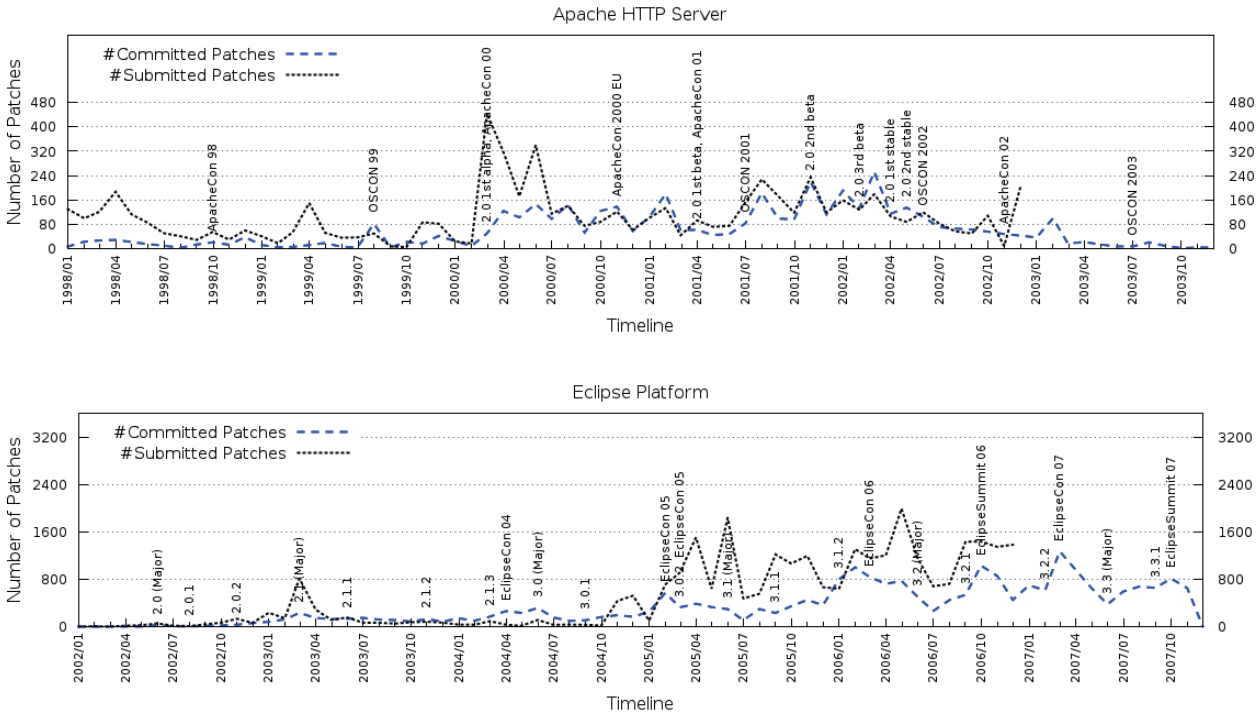


Fig. 11 Comparing submitted and committed patches in Apache HTTP Server and Eclipse Platform.

committed patches was exceeded the number of submitted patches. (i.e. January 2006 in Eclipse Platform) This figure also verifies the arouse number of participants when an event was held. (i.e. March 2000 in Apache HTTP Server, and March 2007 in Eclipse Platform).

6. Conclusion

We have invented a novel algorithm that fills the unexplored area of patch-related activities in OSS study. Our proposed algorithm can identify a portion and gradual patch accepted cases in the patch acceptance analysis. It will lead researchers to a better understanding of OSS project societies. Recently, a study based on patch acceptance analysis has simplified the procedure improperly by omitting from the study both of the important cases mentioned here. This unfortunately led the researchers to a fallacious research conclusion, because the cases omitted always occur. To prove the potential of the algorithm, we constructed a patch analysis framework and perform two case studies using Apache HTTP Server, and Eclipse Platform. They are large and well-known OSS projects.

Our first case study brought many insights into patch acceptance study, which is improved by the utilization of our proposed algorithm. The experimental results from our analyzing framework are very interesting. They include larger size patches and are more likely to be accepted, a conclusion different from that of existing studies. We believe the reason is that the existing coarse-grained algorithm excludes the portion and gradual patch accepted cases that has generated a bias toward larger patches.

In our second case study, we found that exhaustive details of an OSS committer's openness (i.e. the level at which committers accept a suggestion from outside) can be elucidated by extending a temporal-based analysis from our patch acceptance identification algorithm. The analysis reveals two interesting identities of OSS committers' openness. The first is that a special event occurrence in an OSS project would affect the committers' openness level, and the second finding is that there exists a relationship between OSS committer's openness and OSS project evolution.

Both case studies are good examples that show the promise of our proposed algorithm. In continuing with this research, we would like to seek more criteria that have been overlooked or omitted from the existing patch-related activity studies. This will help researchers come to understand OSS societies better.

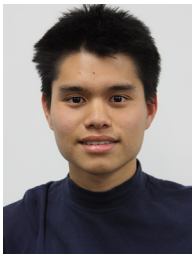
Acknowledgment

The first and third authors are grateful to the internship program cooperated and supported between Kasetsart University, Thailand, and Nara Institute of Science and Technology, Japan. It bestows a grant as well as an opportunity for undergraduate student to achieve a wealth experience in abroad graduated school research.

This research is being conducted as a part of the Next Generation IT Program and Grant-in-aid for Young Scientists (B), 22700033, 2010 by the Ministry of Education, Culture, Sports, Science and Technology, Japan.

References

[1] C. Bird, A. Gourley, and P. Devanbu, "Detecting patch submission and acceptance in OSS projects," Proc. 4th International Workshop on Mining Software Repositories (MSR), pp.26–29, 2007.
[2] B. Sethanandha, "Improving open source software patch contribution process: Methods and tools," Proc. 33rd International Conference on Software Engineering (ICSE), pp.1134–1135, 2011.
[3] G. Jeong, S. Kim, T. Zimmermann, and K. Yi, "Improving code review by predicting reviewers and acceptance of patches," Research on Software Analysis for Error-free Computing Center Tech-Memo (ROSAEC MEMO), 2009.
[4] N. Ducheneaut, "Socialization in an open source software community: A socio-technical analysis," Computer Supported Cooperative Work (CSCW), vol.14, pp.323–368, 2005.
[5] J. Asundi and R. Jayant, "Patch review processes in open source software development communities: A comparative case study," Proc. 40th Annual Hawaii International Conference on System Sciences (HICSS), p.166, 2007.
[6] P. Weißgerber, D. Neu, and S. Diehl, "Small patches get in!," Proc. International Working Conference on Mining Software Repositories (MSR), pp.67–76, 2008.
[7] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye, "Evolution patterns of open-source software systems and communities," Proc. International Workshop on Principles of Software Evolution (IWPSE), pp.76–85, 2002.
[8] A. Bacchelli, M. D'Ambros, and M. Lanza, "Extracting source code from e-mails," Proc. 18th International Conference on Program Comprehension (ICPC), pp.24–33, 2010.
[9] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," Proc. International Working Conference on Mining Software Repositories (MSR), pp.27–30, 2008.
[10] Y.C. Jie Tang, Hang Li, and Z. Tang, "Email data cleaning," Proc. 11th International Conference on Knowledge Discovery in Data Mining (KDD), pp.489–498, 2005.
[11] S. Chakrabarti, M. van den Berg, and B. Dom, "Focused crawling: A new approach to topic-specific web resource discovery," Proc. 8th International Conference on World Wide Web (WWW), pp.1623–1640, 1999.
[12] B. King and I. Provalov, "Cengage learning at trec 2010 session track," Proc. 19th Text REtrieval Conference Proc. (TREC), 2010.
[13] M. Yamamoto, M. Ohira, Y. Kame, S. Matsumoto, and K. Matsumoto, "Temporal changes of the openness of an OSS community: A case study of the apache http server community," Proc. 5th International Conference on Collaboration Technologies (CollabTech), pp.64–65, 2009.
[14] N. Smith and J.F. Ramil, "Agent-based simulation of open source evolution," Software Process Improvement and Practice, pp.423–434, 2006.
[15] A. Capiluppi, J.M. González-Barahona, I. Herraiz, and G. Robles, "Adapting the "staged model for software evolution" to free/libre/open source software," Proc. 9th International Workshop on Principles of Software Wvolution: In Conjunction with the 6th ESEC/FSE Joint Meeting (IWPSE), pp.79–82, 2007.
[16] M.W. Godfrey and Q. Tu, "Evolution in open source software: A case study," Proc. International Conference on Software Maintenance (ICSM), pp.131–142, 2000.
[17] E.S. Raymond, The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary, O'Reilly and Associates, 1999.



Passakorn Phannachitta received B.E. degree in Computer Engineering from Kasetsart University, Thailand in 2011. Now he is currently a M.E. student at Nara Institute of Science and Technology. His research interests include open-source software, data mining, GPUs and cloud computing.

Akinori Ihara received the B.E. degree in Science and Technology from Ryukoku University, Japan in 2007, and the ME degree (2009) and D.E. degree (2012) in information science from Nara Institute of Science and Technology, Japan. He is currently Asistant Professor at Nara Institute of Science and Technology. His research interests include the quantitative evaluation of open source software development process. He is a member of the IEEE and IPSJ.

Pijak Jirapiwong received the B.E. degree in Computer Engineering from Kasetsart University, Thailand in 2011. He participated in an 8 weeks internship program between Kasetsart University and Nara Institute of Science and Technology in Japan from March to April of 2010. He is interested in web mining and search engine.

Masao Ohira received the B.E. degree from Kyoto Institute of Technology, Japan in 1998, M.E. and Ph.D. degrees from Nara Institute of Science and Technology, Japan in 2000 and 2003 respectivery. Dr. Ohira is currently Associate Professor at Wakayama University, Japan. He is interested in supporting knowledge collaboration in software development and supporting use and development of open source software. He is a member of ACM, Human Interface Society, and IPSJ.

Ken-ichi Matsumoto received the B.E., M.E., and Ph.D. degrees in Information and Computer sciences from Osaka University, Japan, in 1985, 1987, 1990, respectively. Dr. Matsumoto is currently a professor in the Graduate School of Information Science at Nara Institute Science and Technology, Japan. His research interests include software measurement and software process. He is a senior member of the IEEE, and a member of the ACM and IPSJ.