2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement

An Analysis of Gradual Patch Application: A Better Explanation of Patch Acceptance

Passakorn Phannachitta*, Pijak Jirapiwong†, Akinori Ihara*, Masao Ohira* and Ken-ichi Matsumoto*

*Graduate School of Information Science,
Nara Institute of Science and Technology, JAPAN
Email: {phannachitta-p, akinori-i, masao, matumoto}@is.naist.jp

†Department of Computer Engineering,
Faculty of Engineering, Kasetsart University, THAILAND
Email: b5005135@ku.ac.th

***Abstract*—Patch submission has been known as one of the most important activities to sustain the open source software (OSS). The patch archive can be analyzed to procure many benefit cognizance for supporting the OSS project works. The recent models and methods that analyze the patches acceptance are quite rack of comprehensive; hence, complex activities such as a committer portioning the submitted patch out and accept are still excluded from the analysis. Therefore, the results derived from those methods would be inadequate to conclude the actual patch acceptance. In this research, we introduce an algorithm for analyzing patch acceptance including the partial and gradually accepted conditions. Validating our algorithm, we present our methods for indicating the partial and gradual application of the submitted patch between either mailing list and SVN or Bugzilla and CVS which are the commonly deployed patch-activities related system. We studied on two well known OSS projects; Apache HTTP and Eclipse Platform. We obtained a fascinating conclusion that larger patches have more confident to be accepted than the smaller contradicted to other analysis that came from the recent methods.**

I. Introduction

It has been concluded that the eminently influential of open source software (OSS) comes from the free willing collaborative effort between a large number of people who forming into a community [14]. There are many activities collaborated between people driving the OSS community. Patch submitting is one of the most crucial activity to sustain each project belonged to the OSS community that makes OSS become a truly community-led development. It provides a major channel for an explicit discussion between everyone who is actively involving with the project. Whenever a user or a developer needs to change some project's components (i.e. source codes, documents, or figures), they just submit their own changes as a patch. The submitted patch will have a further discussion with more people. At last, if the changes in the patch are agreed, it will be accepted by committed into the project repository. That means the changes suggested by that developer will be released with the software in future version. From these activities, we can analyze and procure much benefit cognition in many aspects [5], [7], [11]. In our study, we interested in finding the characteristic of the patches submitting that will have more possibility to be accepted. It will help the developers who are going to submit their patch by giving them some feedback information.

Since the Bird et.at [4] introduced a direct study on patch submitting and acceptance, there have been many following studies tried to discover more explanation about the relationship among the patch-related activities in OSS [1], [6], [7], [12]. The motivation of our study comes from the conclusion of Weiβgerber et.al studied [15]. They believed small patches (i.e. a patch contains small changes) are more preferable to the committer, and it will have more possibility to be accepted. We figure out that their demonstration does not reflect well enough. Since they concluded a patch as accepted if and only if the whole suggested changes are committed to the repository at once. Consequently, the "accept by committed at once" condition will severely decrease tho possibility on large patches to be concluded as accepted. In fact a patch can be accepted just its portion as well as it can be accepted gradually (i.e. a method in each commit). Therefore, more applicable method to consider the size of accepted patch should concern those conditions.

We devise an algorithm to analyze the portion patch acceptance concerning only source-code patches. Evaluating our proposed algorithm, we develop a method to extract patches from mailing list and Bugzilla. Mailing list is usually used for verifying patches, and Bugzilla is for tracking bugs. They are the common channels that developers discuss about patching. After we extracted all patches, we identify the percentage of the acceptance in both fully accepted and partial accepted cases. An analyzing on partial accepted cases differentiate us from Weiβgerber's proposed [15], and the including of gradual accepted cases will make our proposed method unique. The further analysis will tell us an approximate lines of changed code are submitted patches more accepted. In this study, we perform several experiments on Apache HTTP and Eclipse Platform projects. They have used totally different environments. Our delivered results are very interesting that we can confidently conclude that small patches are not more accepted than the large patches, which contradicts with our motivated research.

The remainder of the paper is organized as follows: Section II briefly explains the backgrounds related to the patches submitting activities. Section III introduces our research questions. Section IV explains our proposed algorithm and its implementing method. Section V describes the dataset for our study, explains the experimental setup, and elaborates the experimental results. Section VI discusses on our finding and validating our proposed. Section VII provides conclusion. And finally, section VIII outlines some future works.

II. Backgrounds

A. Repository

Achieving the sustainable community-led development, an open-source project needs to reveal its components to public. There is a common place called repository to store all of the necessary project correspondences (i.e. source code and documents). Each stored file or even the whole repository itself (depends on repository-management software) has a revision number that uses as a checkpoint indicator for any changes committed to the repository. Revision number increases automatically in every commit, and sometimes revision number is also used as a version releasing number.

Nowadays, it's very convenience since there are more tools accessible to the software repository such as web browser, client software, integrated development environment (IDE) tool, and repository-management software. Normally, general users need just simple tools (i.e. web browser or basic client software) to access the repository hence those software provide a plenty of functions covered their needs. For example, they can automatically collect software packages together with the dependencies and install them to their local system using just those simple tools. On the other hand, developers need more functions for their works. In practical, they usually use more powerful tools that can check out any file imprinted with its revision number directly from repository. They can apply for any case study, fix bugs or develop a new component. Recently, there are many well-known repository-management software such as CVS, SVN, and git that provide many features for managing files as well as an accessibility to their metadata such as timestamp. One of the most important features is logging the changed information for every file in the repository. These logs can be analyzed in many aspects for the patch-related activities comprehension.

B. Patches

Patches are widely used in OSS. When a non-committer, who does not have a permission to access the project repository directly, needs to alter some portions of source code for fixing a bug, or appending some new features, patch is an affective approach. They just check out the source code from the repository and change it as they want. Then, they will submit their patch to the committers, who have a higher privilege, through any patch-submitting or bug tracking system. After a discussion, if the committers accord with those changes, they will apply for submitted patch to the target file. (We call a file that the submitted patch needs to fix, as a target file.) At last, the changed target file will be committed to the project repositories. We denote this agreement as the patch is accepted. On the other hand, if none of the patch's component is committed to repository, the patch is rejected.

1) Patch Creation: In practical, a submitted patch is a file that contains only the difference between the original target file and the non-committer's edited version. The difference is indicated line by line showing which lines are removed and which lines are inserted. We usually called a file contains the difference as a diff file, which is later become a submitted as a patch.

The diff files are commonly implemented in 3 distinctive formats. They are Standard diff, Unified diff and Context diff format. Each of them describes the changes in different ways. Nowadays, diff file can be created by many software. The simplest method is using Unix-base command; however, it's can produce results only in Standard diff which lacks of context information [15] especially the lack of timestamp. Context information is a crucial information source for performing a patch-related analysis such as our study in detecting the patch acceptance. Alternately, the two others Unified diff format and Context diff format has sounded implementation of context information section, so they are able to be a source for the further analysis. We decide to omit the analyze of patches in Standard diff same as Weiβgerber et al.'s proposed [15]. They've already concluded that Standard diff are seldom used, which we agreed with them.

For example, when a non-committer modifies source code in //local/m/hi.c as shown in Figure 1 and then he wants to create a patch for applying with //repos/m/hi.c in revision 1.2 They just use a diff command from repository software and get the diff result including header, time stamp and changed information that use as a patch. Figure 2 shows the result for each diff format applying each diff command on cvs repository.

1	void main() {	1	#include <stdio.h>
2	printf("Hi");	2	void main() {
3	}	3	char n[] = ABC;
4	/* Say Hi */	4	printf("Hi %s", n);
		5	}
//repos/m/hi.c (revision 1.2) Last Modified: May 2010 21:01:44		//local/m/hi.c (modified in local) Last Modified: Jun 2010 17:56:08	

Fig. 1. An example on comparing between changed source code and its target file

C. Partial Patch Acceptance

When a non-committer submits his patch to the OSS community, others who are actively joining the community can see and may have some comments discussed on the submitted patch. The discussion will help the committers to decide if they should accept that patch or not. In case a submitted patch is source code, it may be accepted only in portion (i.e., some lines of code). For an example scenario, after a patch has already been discussed, if the committers conclude to accept that patch, but it still has some small defects or it has a room for an improvement, they will modify some portion of

Format	Standard Diff	Unified Diff	Context Diff
Command (CVS)	cvs diff -r 1.2 /m/hi.c	cvs diff -u -r 1.2 /m/hi.c	cvs diff -c -r 1.2 /m/hi.c
Header (CVS)	Index: hi.c ============================= RCS file: /m/hi.c,v retrieving revision 1.2 retrieving revision		
Modified time		— hi.c 28 May 2010 21:01:44 -0000 1.2 +++ hi.c 14 Jun 2010 17:56:08 -0000	

Modified contents	0a1
	> #include <stdio.h>
	2c3,4
	< printf(Hi);
	—
	> char n[] = ABC1;
	> printf(“Hi %s”, n);
	4d5
	< /* Say Hi */
		@@ -1,4 +1,5 @@
		+#include <stdio.h>
		void main () {
		- printf(Hi);
		+ char n[] = ABC;
		+ printf(“Hi %s”, n);
		}
		- /* Say Hi */
			***************
			*** 1,4 ****
			void main () {
			! printf(Hi);
			}
			- /* Say Hi */
			— 1,5 —-
			+#include <stdio.h>
			void main () {
			! char n[] = ABC;
			! printf(“Hi %s”, n);
			}

Fig. 2. Obtained Information form each kind of the diff format

that patch before commit it. Certainly, this patch is accepted. On the contrary, some prior studies concluded this type of situation as rejected [15]. Moreover, size of patch is also related to the chance of the fully acceptance, a large patch may contain several methods that committer can decide to accept only some methods. We called this a gradually accpetance. In order to reflect these situations, we need an analysis of the patch acceptance in portion that include the partial and gradual patch acceptance case. It will give us more precise details such as we can achieve information about the patch acceptance in percentage. For examples, if a patch contained twenty lines of changed code was submitted, then it is committed just fifteen lines of diff code. This patch is accepted by 75%. Further, with this analysis, we can approximate the range of the submitted patch which has a sufficient confidence to be accepted. This information could not be obtained with an analysis concerning only the fully acceptance.

Recently, an analysis in partial and gradual acceptance cases are still nonexistence. In this research, we want to appraise how much benefit will we procure when we finer the grain from the completely whole patch accepted to partial patch accepted. At least, we believe firmly that the results from our analysis method will give the OSS developers or users more comprehensiveness on the patches submitting and acceptance activities beside the result from recent methods.

III. Research Questions

First of all, not only deeper particulars on patch acceptance we gain by the finer measuring, but we also implicitly gain benefit knowledge that we are unable to achieved without analyzing the patch acceptance in partial. Our goal on observing the partial acceptance is to answer two questions:

Q1 We would like to know how large of submitted patches are really more accepted:

In this research we will analyze the patch acceptance using as small grain as the lines of submitted code, and we measure the acceptance in percentage of the partial accepted in each classified length of patches. For example, we will conclude if 5 of 10 changed lines code in the submitted patch are accepted as 50% accepted. It is be more appropriate to conclude a patch as accepted than the recent method that conclude a patch as accept if the whole patch is accepted [15]. Furthermore, when we aggregate all information that we have, we will have sufficient information, and be able to clearly visualize the acceptance rate and lead us to the conclusion the size of patches that are really more accepted.

Q2 How long is the considerable lines of changed code of the submitted patch that have a high confident enough to be accepted:

Finding how much confidence of a submitted patch to be accepted will be very precious. The direct benefit is guiding the non-committer the minimum length of patches should they submit in order to achieve a high enough confidence of an acceptance. There is also an indirect benefit, which we think it is more precious. It will be an excellent and influent feedback to encourage the non-committer to contribute and submit more patch if the acceptance rate is high. Non-committer will feel like they are well-treated by the committers. It also incites the committers to be more open to the non-committers and accept their patches and suggestion more in case the acceptance rate is low. Eventually, It will increase the number of mutual activities that lead the OSS community to have more comprehension between everyone.

IV. Methodology

The main idea of our proposed methodology is demonstrated in the following scenario, and it is illustrated in the Figure 3.

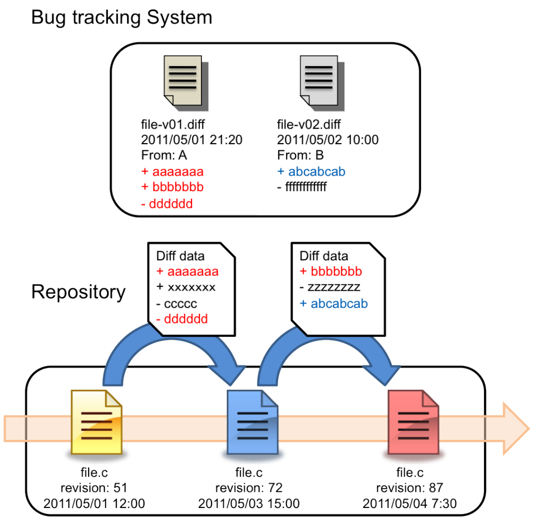

Fig. 3. Outline Scenario

A non-committer 'A' checked out file.c from the repository on 2011/05/01. At that time, the repository's revision number is 51. Then he removed two lines and appended one line of code to that file in his local. To submit these changes, he prepared a file-v01.diff using the diff command between the checked out file and his edited. Then, he submitted it. Not that, this patch has three changed lines of code. At the software repository side, the corresponded target file may be committed several times after that patch is submitted. In this given example, the first adjoining updated on the future revision (rev.72) of the target file has updated four changed lines of code, and there are two corresponded changed lines that are matched with the submitted patch. Its means the committer had accepted two changed lines of code suggested by this non-committer. If the analysis ended here, we will conclude this submitted patch as 66.67% accepted (two of three lines of code is accepted). However, one other line was appeared in next revision (rev.87), so at this point of time this patch is concluded as fully accepted. Next, in file-v02.diff's case, the next adjoining updated revision (rev.72) contained only one corresponding changed line, and there was not any other corresponding changed line in that future revisions, so we will conclude this patch as 50% accepted.

In detail of our implementation, we divided our proposed method into three phases. They are Patch extraction, Diff file creation, and Partial acceptance identification. The first and the second phase is a data preparation. At the Patch extraction phase, since we decided to study on two systems, Mailing list and Bugzilla, which are totally different, we need two specific patch extraction methods. There are several recently proposed methodologies for extracting patches from mailing list [2], [3], [9] . We choose to improvise and extend the Weiβgerber et al.'s proposed [15] . Their proposed method is straightforward that is the most suitable for us since improving a method to extract patches from an email is out our scope. On the other hand, we have to develop a method to extract patches from bug-tracking system from the scratch since such an explicated proposed method is nonexistent. After the patch extraction is finished, we will start the second phase at the software repository side. Our goal on this step is to gather required information in repository that can match with the extracted patches data. Matching them will lead us to obtain the acceptance rate. We also have two types of software repositories: CVS and SVN. For each type of repositories, first, we gather all revision number and timestamp of each file in the repository, and then we dump them to a file called committed log. Note that there is one commit log per each target file. Next, we will detect the difference (lines of changed code) between each adjoining updated revision and dump all the difference into a file called diff file. At the third phase, we will compare each line of changed code between the repository's diff file and the extracted patch. The final result will deliver us the partial patch acceptance information.

A. Patch Extraction

1) Patch extraction from mailing list: At the beginning step, we follow the Weiβgerber et.al's proposed [15] to extract raw-patch from emails. The goal of this phase is to transform those extracted raw-patch data into proper format. There are three kinds of patches that we need to identify at first. From this step we will exclude the standard diff format from our study since it has insufficient information. We then break the rest of Context diff and Unified diff format into readable line of changed code. After that we make records composed all the necessary information. Let a tuple $(I_p, P_p, t_p, L_p, [c_p])$ denotes a patch, where I_p is the patch's index, P_p is the patch's absolute path that we can identify the corresponded target file in the repository, t_p is the patch's submitting time as a timestamp of each patch. It is indicated at the mail header. L_p is the total number of the changed lines of code, and $[c_p]$ is a list contained all the individual changed lines of code. Note that we choose the line of changed source code as our grain since it is the smallest meaningful grain that we can assure the intense committed code.

There is an issue on indicating $(I_p, P_p, t_p, L_p, [c_p])$ from raw patch data. It is about the time zone that affects the timestamp (t_p). We observe our data manually and found that there are many types of time zone indicator such as UTC, EST, and EDT. Without parsing them into a common time zone, it will certainly affect the accuracy of the analysis. Even so this issue has not been discussed in any existing works. In our work, we decide to parse all time zone into UTC. At last, After we finish composing the patches record, we will store them in database.

2) Patch extraction from Bugzilla: We developed a simple web-site crawler for gathering the raw patch data from Bugzilla, because it is deployed in web-base application. The crawled data are in HTML format, which we need to parse them and discover where the raw patch data are located. We use an Html-parsing tool named Jericho HTML parser [8]. It is an outstanding effective HTML parser implemented in Java [10] . Unfortunately we have ascertained that patches are always augmented in the attachment section of Bugzilla, so that we need to enhance our simple web crawler into a specific one dedicated for this task. After we had gathered the attachments, we explored them manually, and we have found that there are many types of the file archive containing the patches. Note that our developed web crawler has already respect to the politeness policy on crawling the Bugzilla.

Since everyone is allowed to submit any type of files to Bugzilla. (i.e., patches, documents, or figures). We have to distinguish the raw patches out of the others. Ideally, if a file contains some patch-indicating messages, we can conclude if that file is a patch easily. However the patch-indicating messages are not widely used, so we need a deepen inspection to find out more patch-indicated characteristics. We develop a heuristic to handle that issue. It works as following. At the first step, we recursively extract all the attach file archives ignoring any non text-base file. (i.e. .jpg and .dll) Then, we scan each text-base file thoroughly, and match them with the patch's keywords. Those are "Index:", "RCS file:", or "diff". At last, we will conclude if an observing file is a patch by finding the name of an existed target file, which the patch needs to fix. Finding the corresponded target file is mandatory because it is impossible to analyze the patch acceptance without that information. From this step we can follow the method in the mailing list patch extraction those are identifying the patch type, excluded the standard diff, clarifying the time stamp, composing them into records and at last store them into database.

B. Diff-file extraction

At the beginning of this phase, we check out all the project's source code from the repository. Then, we create its corresponding committed log linked by its file name. Figure 4 shows an example of our committed log. Each line represents one revision. There are three columns separated by a whitespace in each line. Numbers in the first column indicate the revision number. The second column and the third column indicate the timestamp.

After we had all committed logs of every file in the repository, we create diff file that contain the difference between the adjoining updated revision. In our study, the diff file is in context diff format.

We also treat the diff files as a tuple as the extracted patch same as patches. Let a tuple (I_r, P_r, t_r, $[c_r]$) denotes each diff file. P_r is the committed source code's absolute path, I_r is its index, t_r is its timestamp. We also parse the time zone into a

```
##Rev.No. Committed timestamp
666184    2008-06-11   01:31:59
577830    2007-09-21   02:40:34
106695    2004-11-27   17:23:59
53744     2004-10-05   05:01:25
```

Fig. 4. Committed log file example

common timezone as the patches. $[c_r]$ is the list of changed line in a revision. Note that, the future analyzing steps does not require a revision number because the timestamp can serve us the same propose with more flexibility. We also composed (I_r, P_r, t_r, $[c_r]$) into records and store them in a database.

C. Partial Acceptance Identification

Starting on this phase, we have two databases. They contain patches $(I_p, P_p, t_p, L_p, [c_p])$ and diff files $(I_r, P_r, t_r, [c_r])$. If a patch is accepted, we need to find how much in percentage is it accepted comparing between fully accepted and partially accepted. Further, we need to know in global view that how long in term of line of changed code do the submitted patches tend to be more accepted. We define a time scope Δt to evaluate the acceptance. We will conclude that a $patch_i$ is accepted if and only if there are some lines of the submitted code have been committed with its target file to the repository within Δt. Or we can say that a $patch_i$ is accepted iff:

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

Note that this is more flexible than the fully accepted condition, which is held iff:

$$\begin{aligned} &(I_p = I_r \vee P_p \sim_{match} \; P_r) \\ \wedge \; & t_p + \Delta t \leq t_r \\ \wedge \; & (\forall l \,| l \in [c_p]) \subseteq [c_r] \end{aligned}$$

Ideally, the condition $I_p = I_r$ is sufficient to match between the submitted patch with its corresponded target file. However in practical, we observed and found many records have a duplicate Index or the non-committer has neglected the Index field; therefore, in order to match them, we have to match between their absolute paths (P_r and P_p) instead. Using a heuristic based on the longest string matching with the absolute path; we can identify which target file that the observing patch is belonged to. In case the matching's result return more than one candidate, we will ignore it because of the ambiguity. Finally, the acceptance rate for an individual $patch_i$ is obtained by

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

V. Experiment

We apply our proposed methodology for studying the partial acceptance with two well-known open source software projects, Apache HTTP and Eclipse Platform.

1) Apache HTTP: Apache has been introduced as a community-led development software foundation since 1999. Nowadays, there are many fascinating sub-projects in the name of Apache such as web server, search engine development framework, and distributed computing framework. They are currently deployed around the world. Our study will focused on the very first Apache's sub project named Apache HTTP.

2) Eclipse Platform: Eclipse is a well-known interface development environment (IDE) project. It provided many IDEs those are recently in-use in many programming languages especially the most well known, Java Eclipse. Eclipse project was created in the name of IBM in 2001, and later became an open-source project. Today, Eclipse still uses the same name for its basic IDE. Eclipse also has many additional IDE features as sub projects using name extension from Eclipse for any such as Eclipse Platform, which is our focused in this study.

A. Experimental Setup

Study on two main routines with two OSS projects, at first, we perform a case study in Apache HTTP project between its patch discussing system called mailing list, and its project repository named SVN. The second study routine analyses Eclipse Platform project between its bug-tracking system called Bugzilla and its repository called CVS. In the experimental setup, we prepare data by transforming raw data into records and then composing them into databases using methods that we mentioned in the previous section. In both study routines, we observe the changed line of code in each patch in two aspects. The first aspect is analyzing them as plain texts as they were submitted, but in the second we collapsed all white spaces before perform the analyzing. The assumption behind these two aspects is the committers are able to manually apply the submitted patch for the source code in the repository. It may produce some white spaces shifted that would lead to an inaccuracy analysis. Table I shows the quantity of each patches data source and each repository from two datasets respectively.

TABLE I
THE CHARACTERISTIC OF OUR DATASETS

(a) Repositories

	Apache HTTP	Eclipse Platform
Repository	SVN	CVS
Observing period	1996/01/01 - 2002/12/31	2001/10/01 - 2007/12/31
#File	4,634	49,681
#Changed line	1,686,133	11,519,296

(b) Patches Data Source

	Apache HTTP	Eclipse Platform
BTS	Mailing list	Bugzilla
Observing period	1998/01/01 - 2002/12/31	2001/10/1 - 2007/12/31
#Patch	6,370	42855
#Target file	2,241	28270
#Changed line	171,354	13,086,169

Raw data from Eclipse Platform are about ten times larger than another from Apache HTTP project in both total numbers of file and total numbers of changed line. We will call the Apache HTTP data as a small dataset and Eclipse Platform data as a large dataset. The ratio between target files stored in repository, and the target file that patches intend to fix is quite equal, which is about 1:2.

For answering the first question about the percentage, the acceptance percentage needs to be divided into several classes for the evaluation. We decide to classify them into five classes. The first is 100% accepted which we denote as fully accepted. Then, the others are partial accepted. They are (0%, 25%), [25%, 50%), [50%, 75%), and [75%, 100%) accepted. Next, in order to achieve length of the submitted patches, we also divide the submitted patches into groups by their length. We also classify them into five ranges as following, 1-4 line(s), 5-9 lines, 20-49 lines, 50-199 lines, and 200 lines and more. We denote 1-4 line(s) as a small patch [15]. We also denote the class of 200 changed lines and more as a large patch. For the rest of the range, we select the gap between each range growing in exponential.

B. Experimental results

For the following experiments, we set the time scope (Δt) parameter as 7, 30, and 365 days. From an assumption that a patch could in the sight of the committers and later has an influence to them to decide if it should be accepted only within some time limit. We assume the time scope of 7, 30, and 365 days; hence, they are meaningful thresholds (a week, a month, and a year). In the partial acceptance identification phase, at the first step, we deploy our method on the small dataset, Apache HTTP. Firstly, we test how significant between collapsing and non-collapsing white spaces for each line of changed code. They are expressed in Figure 5 setting Δt = 365 days.

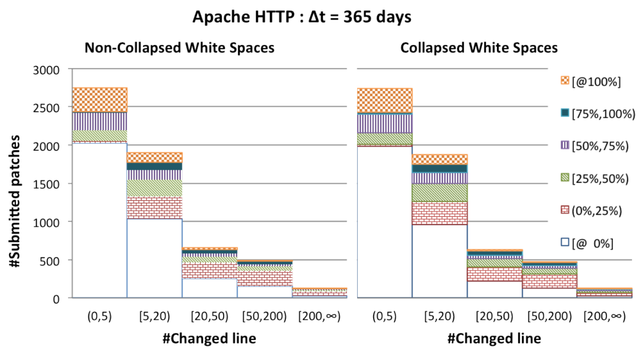

Fig. 5. Comparing between non-collapsed and collapsed white spaces on Apache HTTP dataset. Setting Δt = 356 days

The Y-Axis indicates the total number of submitted patches, and the X-Axis indicates the range of submitted patch that we classified in five classes. Areas filled with different patterns in the each column show the percentage of the acceptance of each class. For example, the left-most bar on the left graph indicates the acceptance of the small patches. There are about 2,750 submitted patches in this range. The area filled with white (The bottom most) shows that 2,000 patches are rejected.

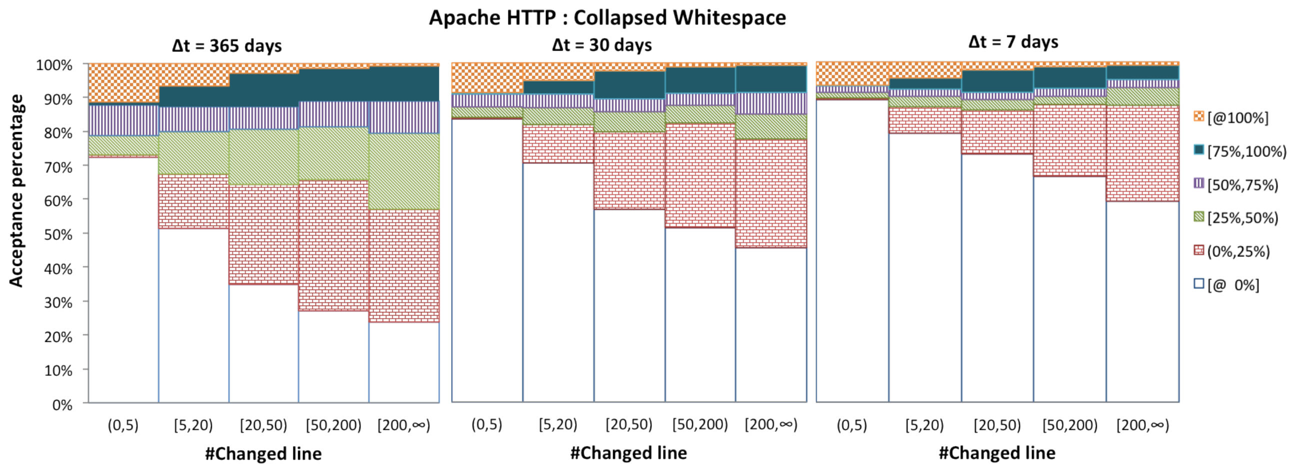

Fig. 7. Acceptance percentage from Apache HTTP dataset

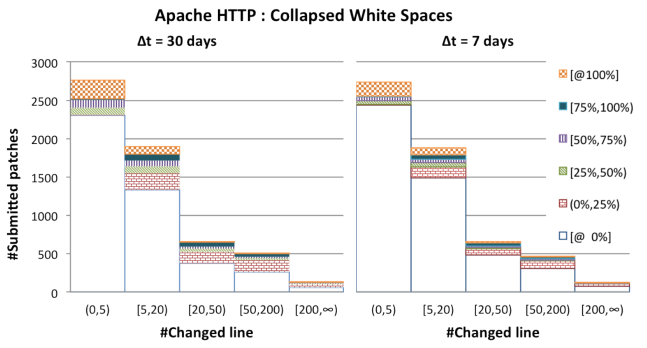

Fig. 6. Comparing between setting Δt = 7 days and 30 days on Apache HTTP dataset. Both are collapsed white spaces

Alternately, the population in the top most area (filled with the checkerboard pattern) in this bar is about 300 patches. This area indicates the 100% accepted. For this case, it's mean 2,650 small patches are submitted and about 300 of them are completely accepted.

The collapsed white spaces delivers slightly better results (i.e. more acceptance rate); however, they are quite insignificant. Besides, both of non-collapsed and collapsed aspect delivers the same trend, in the further results we will illustrate only from the collapsed whitespace. Figure 6 shows the comparing between Δt = 7 days and 30 days only in collapsed white spaces. These preliminary results show that the acceptance rate deceases by the decreasing size of Δt. The different size of Δt will then be discussed after the experiments.

From our small dataset, the non-committers obviously prefer to submit small patches to the large patches. Almost a half of the submitted patches are a small patch. We also found that majority of submitted patches in any class is rejected. It is agreed with several existing works [4], [13], [15]. However, we found an interesting issue from the results that larger submitted patches seem to be more accepted. In order to proof that, we put a rough estimation on the graphs, and Figure 7 illustrates the patch acceptance in percentage, which is more expressly stated in the rate.

The Y-Axis indicates the percentage of the submitted patch. The X-Axis also indicates the range of submitted patch that we classified in five classes same as the previous type of chart. This type of chart displays the result in percentage which is better for concluding the accepted rate. For example, the leftmost column on the left most graph, which is explained with Figure 5, that 300 patches of 2,750 submitted patches are fully accepted. It also means 11% of the submitted patches in this class are fully accepted. Alternately, 2,000 of 2,750 patches that are rejected are about 72% of the population shown in white area.

Figure 7 concludes that longer patches are more accepted. It is contradictory to the proposed of Weiβgerber et.al in small patches get in! [15], since their tests concern only the fully accepted. Likewise, the @100% areas in Figure 7 that represent the fully accepted from every Δt are also decreased by the larger size of the patch.

Next, we perform further analysis with our large dataset, Eclipse Platform. We use the same varieties of Δt time scope as 7, 30, and 365 days. Their corresponding results with collapsing whitespace are reported in following the Figure 8.

For instance, we observe that Eclipse Platform has less acceptance rate than Apache HTTP. We are pretty sure that is caused by Eclipse Platform has about ten times more on candidates comparing to Apache HTTP. At the patch submitted histogram, it's different from Apache HTTP that quite larger patches are more submitted. However, accepted rate's trend in percentage is as same as Apache HTTP's.

VI. Discussion

First of all, we would like to express that our proposed method behind a sounder hypothesis delivers a contradictory result to the recent research [15]. As we explained in the

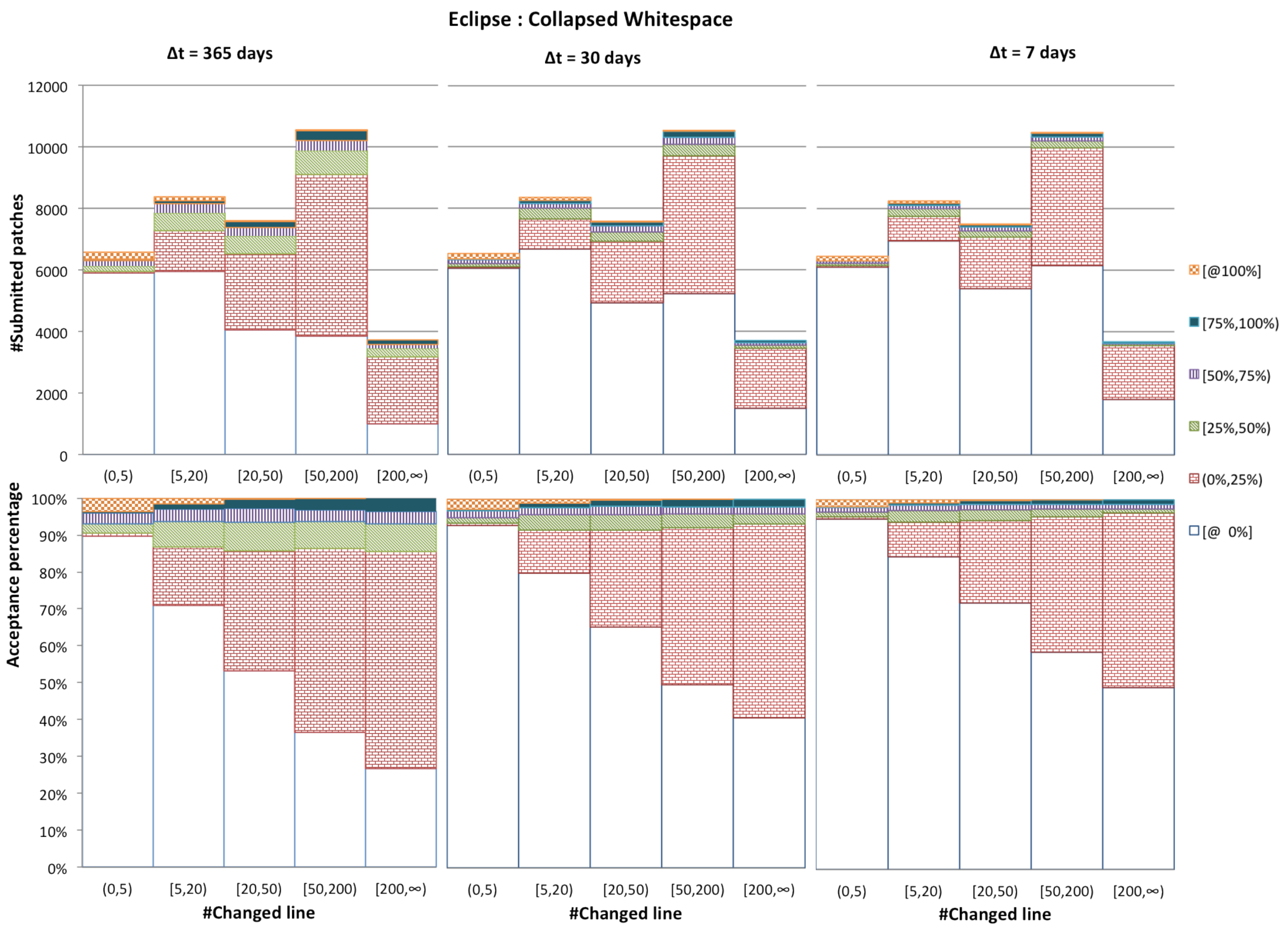

Fig. 8. Experimental results from Eclipse dataset

previous section that the patch acceptance results analyzing from a rough grain analysis wont't be reflected well enough. The following of this section will discuss on what we have found in this work and express the confidence of our proposed.

A. *What do we gain from the varies of time scope?*

Comparing with both Bird et.al and our motivated research, Weiβgerber et.al [4], [15], the results derived from our method are more superior. Since their proposed compared only between a patch and one repository file only in single revision. It is different from us that we compare between a patch and repository files through the timeline until it is fully accepted or it reaches the time limit. This yields more reflective to the actual activities since a patch can be accepted gradually. Another interesting topic is the longer time scope leads us to more acceptance rate besides it still respects with the patch acceptance trend (i.e. Δt = 365 days in each experiment delivers up to two times of accepted rate comparing to Δt = 7 days). Since we certainly aware of the duplicate counting by ensured that we count each changed line of code just once, we believe the more acceptance rate that we gained is sounded. In addition, the different gap on larger patches is more significant. It also concludes that a patch could be more gradually accepted. The sounded reason that the different time scope affects more on larger patches is a large patch would have higher possibility to contain more components. Conclude that the more components it has, the more dependencies it needs to clarify. That's why comparing it with only one revision sounds not a good idea.

B. *Situations to accept the patch in partial*

There are two main situations that can explain the partial accepted. The first is the committer accepted the submitted patch without making any edited. The second one is the committers use the submitted patch as an outline. Then, they code the new patches by themselves and committed. The first situation can be fully detected using our methods. Alternatively, only the approximation approach can handle the second situation. For example, Bird et.al [4] tried to justify that patch submitted paradigm but only the simple cases can be handled (i.e. they can detect if some parts of a variable have been changed.) However, if the whole source code has been refined, their approach would not work. In our work, we did not develop a method to handle those situations. The result that compares between the collapsed and non-collapsed whitespace, which concluded as their differences are quite insignificant, can explain that it tends to be just few occurrences on the second patch submitting paradigm. The difference between collapsed and non-collapsed whitespace should have been much higher, if the committers usually accept patches using their method mentioned on the second situation.

C. Elaboration on the size of accepted patch

Our results concluded the contradictory that the lager patches have more acceptance rate. In this subsection, we would like to discuss on the length of changed line of code. We've given some explanations in the previous section that we distribute them with the exponential gap growing from four lines until 200 changed lines of code. They are also having their meanings. The short patches, which are below five lines, would be just a little changed, collecting some errors. It is the most particular size submitted to Apache HTTP. Resulting from the highest probability for the fully accepted, Weiβgerber et.al concluded this type of patches has the most acceptance rate [15] . The length between [5,20) changed lines of code is the average length of interface or method modification. [20,50) and [50,200) lines seem to be the component modification, and error correction with has greatly large dependencies respectively. For the largest size of patches, we believe it will be for fixing such a severe error, error correcting on such a huge component, or suggesting a totally new component. From our results, the fully accepted area decreases by the larger size of submitted patches. The fully accepted of the larger patch is rare since the submitted patch must be one that is perfectly function with all the dependencies. Since more than one hundred line of code must be accepted in order to conclude the 50% accepted in any large patch, it make us believed our test methods particularly prone to noises. So we have a high confidence to believe that our experimental results are valid.

D. Acceptance confidence when submitting a patch

Answering our second research question about the proper length for the submitted patch, the experimental results figures let us know briefly that the longer patch has higher confidence to be accepted. At first, we figured out that it is depended on projects to conclude the minimum length of submitted patches that have more than 50% acceptance confidence. (i.e. Half of submitted patches in this length are rejected) Submitting a patch to Apache HTTP will achieve that confidence since a very small length of five changed lines of code. On the other hand, submitting a patch to Eclipse Platform needs more than twenty changed lines of code in order to achieve that confidence. It is obvious to conclude committers in Eclipse platform has less openness comparing with Apache HTTP. If this analyzed figure become a common indicator for a cross project evaluation, it will encourage non-committers to have more contribute to Apache HTTP, as well as incite the Eclipse Platform's committer to accept more patches. However, our result guarantees that the larger patch non-committer submitted the higher confidence that the patch will be accepted.

VII. Conclusion

In this research, we introduced an algorithm for analyzing the patches' partial acceptance. It is more suitable to conclude the patch acceptance characteristic than the recently proposed [15] that concluded the small patches are more accepted by judging a patch is accepted only on when the whole patch is accepted. However, a large patch may contain many components that could be accepted gradually in actual.

Evaluating with our algorithm, we construct a method, and perform two study cases with two well-known OSS project. They are Apache HTTP and Eclipse Platform, which we treat as a small dataset and large dataset respectively. We have two types of patches' data sources to extract parches; mailing list and Bugzilla. Extracting patches from mailing list we have improvised Weiβgerber's techniques [15] and extended them with our heuristics. Alternatively, for the Bugzilla, since an explicitly proposed technique is nonexistence, we have to develop our heuristic-augmented website crawler for extracting patches from Bugzilla. We will ascertain the application of the extracted patch in the project repositories line by line, which we believed it's a proper grain. We also assume the time scope when a patch is valid through the sight of a committer.

We procure many interesting results from our partial patches acceptance analysis. Both of our dataset delivered us that the rate of patch acceptance increases by the length of patches size. We also found that the confidence in achieving a high confidence in submitted a patch and it will be accept is depended on the project. The results from the vary of our defined patch analyzing time scope have assured that a patch could be gradually accepted, since larger patches composed with more components have more affected from the longer time scope.

VIII. Future Work

Our proposed can identify a submitted patch that had some minor refinement by the committer before it was committed. Incidentally, if the committer intended to refine and alter all parts of the submitted patch. Recently, there is still nonexistence approach to include this situation into the analysis. It is very interesting if we can achieve those explanations. They will lead us to the basis of patch acceptance characteristic, so that the acceptance rate results derived from the changed lines of code will be perfectly sounded and reflected.

Beyond our proposed in this research, which analyzed the patch acceptance in line of code. In the next research, we will explore more aspect and analyze the other features of code. For example, we will discover how influence of the number of class, code coverage, and cyclomatic complexity will affect the acceptance rate. Those information will be a broader perspective in analyzing patch submitted and acceptance, since the committer may be concern on those attributes in order to accept a patch.

Acknowledgment

The first and second 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 work was conducted as part of StagE Project (the Development of Next Generation IT Infrastructure), Grant-in-Aid for Scientific Research (B), 23300009, 2011, and Grant-in-Aid for Young Scientists (B), 22700033, 2011 by the Ministry of Education, Culture, Sports, Science and Technology, Japan.

References

[1] J. Asundi and R. Jayant, “Patch review processes in open source software development communities: A comparative case study,” in *Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS ’07)*, 2007.

[2] A. Bacchelli, M. D’Ambros, and M. Lanza, “Extracting source code from e-mails,” in *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension (ICPC ’10)*, 2010.

[3] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, “Extracting structural information from bug reports,” in *Proceedings of the 2008 international working conference on Mining software repositories (MSR ’08)*, 2008.

[4] C. Bird, A. Gourley, and P. Devanbu, “Detecting patch submission and acceptance in oss projects,” in *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR ’07)*, May 2007.

[5] C. S. Corley, N. A. Kraft, L. H. Etzkorn, and S. K. Lukins, “Recovering traceability links between source code and fixed bugs via patch analysis,” in *Proceeding of the 6th international workshop on Traceability in emerging forms of software engineering (TEFSE ’11)*, 2011.

[6] N. Ducheneaut, “Socialization in an open source software community: A socio-technical analysis,” *Computer Supported Cooperative Work (CSCW)d*, 2005.

[7] 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 2009-006)*, 2009.

[8] Jericho html parser. [Online]. Available: http://jericho.htmlparser.net

[9] Y. C. Jie Tang, Hang Li and Z. Tang, “Email data cleaning,” in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining (KDD ’05)*, 2005.

[10] B. King and I. Provalov, “Cengage learning at trec 2010 session track,” in *The Nineteenth Text REtrieval Conference Proceedings (TREC 2010)*, 2010.

[11] P. C. Rigby and M.-A. Storey, “Understanding broadcast based peer review on open source software projects,” in *Proceeding of the 33rd international conference on Software engineering (ICSE ’11)*, 2011.

[12] B. Sethanandha, “Improving open source software patch contribution process: methods and tools,” in *Proceeding of the 33rd international conference on Software engineering (ICSE ’11)*, 2011.

[13] B. Sethanandha, B. Massey, and W. Jones, “Managing open source contributions for software project sustainability,” in *Proceedings of Technology Management for Global Economic Growth (PICMET ’10)*, July 2010.

[14] A. M. St. Laurent, *Understanding Open Source and Free Software Licensing*. O’Reilly Media, 2004.

[15] P. Weißgerber, D. Neu, and S. Diehl, “Small patches get in!” in *Proceedings of the 2008 international working conference on Mining software repositories (MSR ’08)*, 2008.