2012 28th IEEE International Conference on Software Maintenance (ICSM)

How Often Do Unintended Inconsistencies Happen? — Deriving Modification Patterns and Detecting Overlooked Code Fragments —

Yoshiki Higo and Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University,
1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan
Email: {higo,kusumoto}@ist.osaka-u.ac.jp

***Abstract*—It is difficult to keep consistent source code. Unintended inconsistencies occur unless we recognize all the code fragments that need to modify in a given bug fix or functional addition. Before modifying source code, keyword-based search tools like grep or code clone detection tools can be used to prevent code fragments from being overlooked. However, once inconsistencies occur in the source code, such tools cannot help us adequately. In this paper, we propose a new method to identify unintended inconsistencies in source code automatically. The proposed method analyzes source code modifications in a repository to derive modification patterns. A modification pattern indicates what kind of code and how it was modified. The derived modification patterns are queries to identify unintended inconsistencies from the latest version of source files. We implemented the proposed method and applied it to FreeBSD and Apache HTTPD. As a result, we identified many overlooked code fragments for bug fixes, functional enhancements, and refactorings. The precisions were 73.4% and 88.9% for the two systems, respectively.**

***Keywords*-Inconsistency detection, Modification patterns, Static analysis**

I. Introduction

In order to complete a task of source code modification like bug fix or functional enhancement, similar code fragments that needs to be changed in a similar/same way should be recognized. If we overlook one or more such code fragments, the modification raises unintended inconsistencies. The overlooked code fragments become factors of faults in the future, which means they not only reduce the availability of the system but also require further modification cost.

When we identify a code fragment to be modified and we do not modify it yet, keyword-based search tools such as grep or code clone detection tools will be a considerable help to identify other code fragments to be modified simultaneously. For example, Higo et al. applied grep and a token-based code clone detection tool, CCFinder [1] to an actual modification task for fixing a buffer overflow problem [2]. The result showed that both tools are useful for preventing code fragments from being overlooked.

However, these techniques are not suited for identifying already existing inconsistencies. We need keywords representing inconsistencies for using grep, still we have no way to get such keywords for inconsistencies already existing because we do not know where inconsistencies exist. Li et al. developed a code clone detection tool, CP-Miner [3] for inconsistencies identification. Although CP-Miner detects inconsistencies such as including wrong variables, other kinds of inconsistences such as including wrong statements or missing necessary statements are not detected. Moreover, code clone detection techniques detect a code chunk as a code clone, so that only variable inconsistencies in detected code chunks are detected. If code surrounding inconsistencies is not duplicated to its correspondents, they will not be detected by code clone detection tools. For these reasons, existing code clone detection techniques are not sufficient to be applied to detect overlooked code fragments.

In this paper, we propose a new method identifying inconsistencies automatically. The proposed method firstly analyzes past revisions of a target software system and extracts modification patterns hidden in its evolution. Then, it scans the target version of the system and detects overlooked code fragments by using the patterns. The proposed method does not suffer from the mentioned problems, and it has the following features:

- it can detect not only variable-level inconsistencies but also statement-level inconsistencies,
- it can detect inconsistencies even if they are not in duplicated code chunk,
- it does not require deep analysis of source code, and so it is easy to apply it to many programming languages,
- it has high scalability, less than an hour is required for analyzing million lines of code and its history.

This paper revealed the following facts by experiments on two open source software systems.

- Most of past revisions contain unintended inconsistencies. They are finally modified in later revisions and kept consistent again.
- A revision of a system contains many unintended inconsistencies, which arose from modifications of bug fixes, functional enhancements, and refactoring.

The remainder of this paper is organized as follows: Section II shows an actual example to motivate this research; Section III presents some previous studies related to this research; Section IV proposes a new method to tackle the problem shown in Section II; Section V describes some

```
  :
 354
 355  for (lr = head_listener; lr ; lr = lr->next) {
 356      ap_get_os_sock(lr->sd, &nsd);
 357      if (FD_ISSET(nsd, main_fds)) {
 358          head_listener = lr->next;
  :
 380      num_listeners++;
 381      if (lr->sd != NULL) {
 382          ap_get_os_sock(lr->sd, &nsd);
 383          FD_SET(nsd, &listenfds);
 384          if (listenmaxfd == INVALID_SOCKET || nsd > listenmaxfd) {
  :
1083      /* Associate each listener with the completion port */
1084      for (lr = ap_listeners; lr != NULL; lr = lr->next) {
1085          ap_get_os_sock(lr->sd, &nsd);
1086          CreateIoCompletionPort((HANDLE) nsd, //(HANDLE)lr->fd,
1087                                 AcceptExCompPort,
  :
1313          ap_log_error(APLOG_MARK, APLOG_NOERRNO | APLOG_INFO, server_conf,
1314          "Parent: Duplicating socket %d and sending it to child process %d.", lr->sd );
1315          ap_get_os_sock(lr->sd,&nsd);
1316          if (WSADuplicateSocket(nsd,
  :
```

Figure 1. server/mpm/winnt/mpm_winnt.c (revision: 83,955)

implementation details of the proposed method; Section VI shows evaluation results on two open source software systems; Section VII discusses the validity of the evaluation results; Section VIII concludes this paper.

II. Research Motivation

Figure 1 is a source file of the Apache HTTP project. The 356th, 382th, 1,085th, 1,315th lines include the same instruction. Three of them (the 356th, 382th, and 1,085th lines) were changed to `ap_get_os_sock($nsd, lr->sd)` in revision 83,956. Then, in revision 83,960 the remaining 1,315th line was modified and the 4 lines code are consistent again. The commit log of the revision 83,960 describes that the revision is for modifying the overlooked place in revision 83,956. These evidences show that: the 4 lines must be modified simultaneously; however a programmer overlooked the 1,315th line. This is a typical example of *late propagation* [4], [5]. As shown in this example, a modification task sometimes requires simultaneous changes on multiple code fragments. Such tasks involve the risk of overlooking.

Consequently, in this paper, we formulate the following research question:

Research Question 1: How often do *overlooked and delayed modifications* happen?

When we find a code fragment to be modified, we will identify other code fragments to be modified simultaneously by using grep. Using grep prevents code fragments from being overlooked. However, once inconsistencies have occurred, grep is not suited for identifying the inconsistencies.

Li et al. proposed a method to detect inconsistencies with code clone detection techniques and developed a software tool, CP-Miner [3]. CP-Miner firstly detects code clones, and then investigates variables in the code clones. If a pair of code clones has unmatched variables, the pair is reported as a candidate of buggy code. This method identifies token-level inconsistencies in code clones.

On the other hand, the proposed method identifies inconsistencies based on past code modifications. In the case of Figure 1, The 356th, 382th, 1,085th, and 1,315th lines must be modified simultaneously, still their prior and subsequent code is not duplicated. That is, every of the 4 lines is a single line code clone. Detecting single line code clones with code clone detection technique is unrealistic. We can detect such small code clones by decreasing the minimum size of detected code clones. However, we will get an enormous amount of code clones even if the system is not so large. Extracting necessary code clones from a huge amount of ones is a complicated and burdensome task. Moreover, the bigger inconsistencies are, the more difficult it is to identify them by code clone detection approaches. For example, statement-level inconsistencies are more difficult to be identified than token-level ones.

In this paper, we propose a new method to detect where inconsistencies exist and how they must be modified. The proposed method is free from the above problems. Also, the proposed method can be realized as a fully-automated processing. The proposed method accumulates how source code was modified in the past, and derives modification patterns. For example, by mining modifications between revision 83,955 and 83,956, the proposed method finds that the following change happens 3 times:

```
ap_get_os_sock(lr->sd, &nsd);
            ↓
ap_get_os_sock(&nsd, lr->sd);
```

The proposed method treats frequent modifications as modification patterns. Modification patterns are used for identifying overlooked code fragments. In the case of Figure 1, by using the above modification pattern, we can automatically suggest that the 1,315th line must be changed to `ap_get_os_sock(&nsd, lr->sd);`. In order to evaluate the usefulness of the proposed method, we formulated the following research question:

Research Question 2: How many overlooked code fragments can be detected by the mining-based method?

III. Related Work

Code clone detection techniques can be used for identifying inconsistencies in source code. Detection tools normalize user-defined identifiers in order to ignore their differences. Hence, code clones including token-level inconsistencies are detected. Such code clones are candidates of buggy code.

Li et al. developed a token-based code clone detection tool, CP-Miner [3], which has a functionality to identify buggy code clones. The recommendations of buggy code clones are performed by checking variables correspondence in a pair of code clones. For example, assuming that code fragment A is a code clone of code fragment B. In A, variable x is referenced three times whereas variable y is referenced twice and variable x is referenced once in B. In this situation, CP-Miner judges that, B is generated by a copy-and-paste operation from A and one of x reference was overlooked to be modified. They applied CP-Miner to large scale open source systems such as Linux and FreeBSD, so that they found 87 bugs in detected code clones.

Higo et al. developed a wrap tool of CCFinder [1] for detecting only code clones related to a given code fragment [2]. When a programmer detects a code fragment to be modified for fixing a given bug, he/she wants only code fragments similar to the detected one. In such a situation, detecting whole the code clones in a system is overkill and time consuming. The wrapping tool is a reasonable solution to obtain necessary code clones. They evaluated the tool by applying it to an actual modification task for fixing a buffer overflow problem. The tool could identify almost all the code fragments to be modified simultaneously.

Bazrafshan et al. also developed a tool for identifying code fragments that are similar to a given code fragment [6]. They realized an approximate code search with the algorithm developed by Chang and Lawler [7]. Both the approaches of Higo et al. and Bazrafshan et al. are effective if we know a code fragment to be modified.

Göde and Koschke investigated how code clones had been modified in software evolution on 3 open source software [8]. Their experiment revealed the followings:

- most code clones were modified only one time at most,
- 14.8% modifications on code clones raised unintended inconsistencies, and
- only 3% modifications on code clones introduced bugs.

Rahman et al. investigated relationships between code clones and bugs [9]. They identified code fragments modified for fixing bugs in the past, then checked how much they were code clones. They revealed that more than 80% bug fixes were performed on code fragments that do not contain code clones at all.

Bettenburg et al. investigated inconsistent changes in code clones at release level [10]. Not all the code clones exist in source code over a long period [11], which is an evidence that revision-based investigation is too fine-grained. It unintentionally considers inconsistencies in short-living code clones. They address that it is important to investigate inconsistencies between code clones appearing in release level, which is a version of software that users use. They used clone region descriptor[12] to tracking code clones between versions. Tracking code clones between versions is much harder than between revisions because consecutive two versions include much more differences than consecutive two revisions. Their investigation revealed that only 4% of code clones are factors of bugs related to inconsistencies.

As described in Section II, unintended inconsistencies can occur in code fragments that are difficult to be detected as code clones. The experiment in this paper is a complemented study of Göde and Koschke, Rahman et al., and Buttenburg et al. experimental results.

Li and Zhou proposed a method to detect violations of coding rules from source code and developed a tool, PR-Miner [13]. PR-Miner uses the frequent itemset mining technique to detect a sequence of method calls that frequently appears. Their method requires only a single version of source code, which is the target of violation detection. Their method is tailored to detect inconsistencies on method calls and their related program elements such as variables of actual parameters. On the other hand, the proposed method requires past revisions of the target source files, but it can detect any kinds of violations if the same violations were fixed in the past.

Livshits and Zimmermann also proposed a method to detect coding patterns and error patterns [14]. In their method, firstly, past revisions of the target system were analyzed to extract method calls that were added simultaneously. In their method, a set of the method calls is a coding pattern. Then, dynamic analysis is performed to check whether the coding patterns are included in the execution traces. If included, the coding pattern is regarded as useful.

IV. Proposed Method

Herein, we proposed a new method to tackle the problem shown in Section II. Firstly, we describe an overview of the proposed method (Subsection IV-A), and secondly we define some terminologies used in the proposed method (Subsection IV-B). Then, we explain the details of two processings of the proposed method (Subsections IV-C and IV-D).

A. Overview

This paper proposes a new method that automatically suggests code fragments to be modified. It consists of two processings: one is **mining processing** that analyzes historical information to derive modification patterns; the other is **detection processing** that identifies code fragments to be modified. In the remainder of this section, firstly Section IV-B defines some terminologies that are used in the proposed method. Then Sections IV-C and IV-D explain *mining processing* and *detection processing*, respectively. Note that, we assume that the proposed method is for software systems managed with version control systems such as CVS or Subversion.

B. Definition

Firstly, we define **code fragment**.

Definition 1 (Code Fragment) *A character sequence of a code chunk (one or more lines of code) in source code. Its length is 0 or more.*

Then, we define **modification pattern**, which represents that a code fragment is changed to another one.

Definition 2 (modification pattern) *Assume that, a code fragment before modification is cf_{before}, a code fragment after modification is cf_{after}, and the modification occurred in revision r, then, modification pattern is formalized with a tuple $(cf_{before}, cf_{after}, r)$. In this tuple, cf_{before} must be different from cf_{after}. That is, no modification pattern occurs if the cf_{before} of a modification is textually identical to the cf_{after} of the modification.*

Next, we define two quantitative metrics to prioritize modification patterns. The first one is **support**.

Definition 3 (Support) *This is a numerical metric to represent the number of equivalent modification patterns of a given pattern plus one. Herein, two modification patterns, $mp_1 = (cf_{1before}, cf_{1after}, r_1)$ and $mp_2 = (cf_{2before}, cf_{2after}, r_2)$, are equivalent if the following two conditions are satisfied:*

- *$cf_{1before}$ is textually identical to $cf_{2before}$,*
- *cf_{1after} is textually identical to cf_{2after}.*

The other one is **confidence**.

Definition 4 (Confidence) *This is a probability that a given code fragment (CF) is changed to another code fragment (CF'). Confidence is represented with a fraction, $\frac{m}{n}$. Herein, n is the number of modification patterns whose cf_{before} are CF, and m is the number of modification patterns whose cf_{before} are CF and whose cf_{after} are CF'. The latter modification patterns is a subset of the former ones, so that $m \leq n$ is always satisfied.*

C. Mining Processing

Firstly, we describe the input and output.

Input: source code repository of the target system,

Output: a set of modification patterns with support and confidence values.

The mining process consists of three steps:

STEP1: identifies revisions where at least one source file was modified,

STEP2: extracts modification patterns between every of two consecutive revisions,

STEP3: quantifies modification patterns by calculating *support* and *confidence*.

In STEP1, the mining processing identifies revisions where one or more source files were modified. Source code repository contains not only source files but also other kinds of files such as manual or copyright files. There are revisions that no source files are modified. Consequently, the purpose of STEP1 is eliminating revisions to be ignored because we focus on only modifications on source files.

In STEP2, the mining processing extracts modification patterns from every of two consecutive revisions identified in STEP1. If we obtain n revisions, $\{r_1, r_2, \cdots, r_n\}$ in STEP1, we extract modification patterns between r_1 and r_2, r_2 and r_3, $\cdots$, r_{n-1} and r_n. Modification pattern extraction is performed by using unix diff command.

In STEP3, the mining processing quantifies the extracted modification patterns by calculating *support* and *confidence*.

D. Detection Processing

The inputs of the detection processing are the followings:

- source files of a revision of a target system,
- a set of modification patterns with their *support* and *confidence* values,
- thresholds of *support*, *confidence*, and *place*.

The output are locations of overlooked code fragments and ways to modify them.

If a code fragment is matched in many places of the system, the matched places are unlikely to be overlooked code fragments. The authors think that the number of overlooked code fragments is usually a small value like 1 or 2. Consequently, we introduce a threshold to specify an upper limit of matched places. In the detection process, if a code fragment is matched more often than the *place* threshold, the matched places are not output.

The detection processing consists of the following steps:

STEP1: selects modification patterns,

STEP2: detects overlooked code fragments.

In STEP1, the detection processing selects modification patterns to be used for detecting overlooked code fragments with the thresholds of *support* and *confidence*. If both the *support* and *confidence* values of a given modification pattern are equal to or greater than the thresholds, it is used for detecting overlooked code fragments.

In STEP2, every source file is checked whether it contains any of selected modification patterns. Before checking, source files are normalized with the same rules used in the mining processing, which means that a long character sequence is generated from each file. Assuming that there are a source file f in a revision r_1 and a selected modification pattern $(cf_{before}, cf_{after}, r_2)$. If the character sequence of f contains cf_{before}, the proposed method regards that f has an overlooked code fragment and output the following information:

- location of the matched code fragments (file path, start line, and end line),
- cf_{after}, which is a suggestion how the code fragments should be modified.

V. Implementation

We have implemented the proposed method as a tool. Currently, the tool handles only Subversion repositories. The tool includes both the mining and detection processing functionalities. The inputs of the system are the followings:

- source code repository of a target system,
- a revision number that overlooked code fragments are searched,
- thresholds of *support*, *confidence*, and *place*.

It outputs information related to detected code fragments (see Section IV-D).

We use SVNKit for operating Subversion repositories. In order to realize the mining and detection processings, we use 'svn log' and 'svn diff' commands. In the mining processing, the tool normalizes source code for discarding trivial modifications. Sometimes differences between two revisions include trivial modifications, such as changing indents, adding/removing white spaces, moving braces the next/previous line. Considering such trivial modifications extracts trivial modification patterns, so that the tool yields false positives. To avoid generating false positives, we derive modification patterns in the following way:

1) obtains all the differences between two revisions by using diff command,
2) normalizes the code fragments in diff outputs. The normalization includes deletion of white space, tabs, and new-line characters,
3) extract a modification pattern from a given diff output if the normalized code fragment before modification is different from the one after normalization.

By normalizing and checking the code fragments in diff output, we can avoid generating trivial modification patterns.

In the detection processing, the tool performs a special handling for modification patterns that the code fragments before modification is entirely included in the code fragments after modification. Figure 2 shows such an example.

```
-   cam_release_devq(done_ccb->ccb_h.path,
+   if ((done_ccb->ccb_h.status & CAM_DEV_QFRZN) != 0)
+     cam_release_devq(done_ccb->ccb_h.path,
```

Figure 2. An example that code fragment before modification is completely included in code fragment after modification. This modification pattern is for fixing a bug and it occurred in 3 places between revisions 199,278 and 199,279.

In the example, an if-statement for executing a method call is added. If we simply use this modification pattern for detecting overlooked code fragments, we will unintentionally find all `cam_release_devq` calls even if they already have the if-statements. Detecting all of those instances yields many false positives. Consequently, after the implemented tool detects places matching with code fragments before modification of modification patterns, it checks whether each of the detected places is surrounded by the code fragments after modification of the modification patterns or not. If it is surrounded, the tool does not output it as an overlooked code fragment.

VI. Evaluation

We selected two open source software systems as experimental targets. The overview of the targets is shown in Table I. The first target is Apache HTTPD. The target period is before creating branch *1.3.x*. The *1.3.x* branch is the first one in the development of HTTPD. In other words, before creating branch *1.3.x*, new features additions and bug fixes for releasing version *1.3.0* were performed.

The second target is FreeBSD Kernel. The target period is *9-CURRENT* kernel. The period started by making the branch of *8-STABLE*, and it ended by making the branch of *9-STABLE*. In the period of *9-CURRENT*, a variety of new functionalities were added for FreeBSD *9.0-RELEASE*. In the closing span of the period, new functionality addition is prohibited and developers concentrated on making the source code stable. The period includes 7,689 revisions where at least one .c file was modified, and the revisions are the target of this experiment. For the two systems, we targeted only .c source files, however it is possible to treat .h header files.

In this experiment, we specified the following thresholds:

- *support*: 3
- *confidence*: 1.0
- *place*: 1

As a result, we obtained 737 and 983 modification patterns from HTTPD and FreeBSD, respectively. The execution time were 4 minutes 58 seconds for HTTPD and 12 minutes 16 seconds for FreeBSD. The remainder of this section describes the investigation results of RQ1/RQ2 with those patterns and comparison results with clone detection tools.

A. Investigation for RQ1

We found that 145 modification patterns out of 737 on HTTPD and 94 out of 983 on FreeBSD appeared in two or

Table I
OVERVIEW OF TARGET SOFTWARE

Software	Start revision (date)	End revision (date)	# of target revisions	LOC of end revision
HTTPD	81,442 (1998-06-02 07:18:44)	90,607 (2001-08-24 11:30:27)	2,784	131,675
FreeBSD	196,121 (2009-08-12 19:44:13)	225,533 (2011-09-14 00:57:29)	7,689	3,570,021

more revisions. We carefully checked every of the 145 and 94 modification patterns one-by-one and classified them into the 4 categories as shown in Table II[1]. The columns of the table show the followings:

- "Bug": the number of code fragments where bug fixes must be applied,
- "Refactoring": the number of code fragments where refactoring should be applied. Herein, refactoring includes very simple operations such as changing variable names or adding or deleting braces of if-statement,
- "Enhancement": the number of code fragments that functionality enhancements or expansions must be applied to,
- "Comment": the number of comments where consistencies must be maintained with other comments. This is mainly for changing copyright statements.

We found that 42 modification patterns were bug fixes. The presence of them indicates that developers were not aware of all the code fragments to be fixed. When a developer committed a revision where such a pattern firstly appeared, he was misled into thinking that he had completely fixed the problem. However, that was not true. The developer himself or another one subsequently would recognize that the program behaves in the wrong way. He would investigate the factor of the wrong behavior, and he would find out that the previous fixes were not enough and there is one or more code fragments to be fixed in the same way.

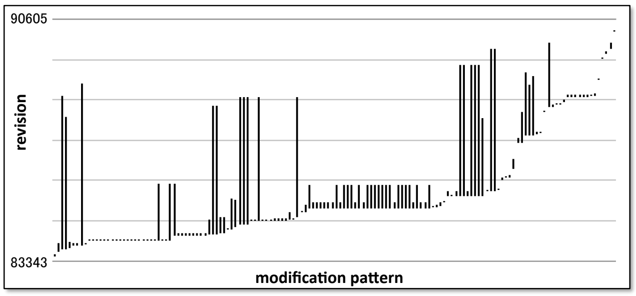
Unintended inconsistencies by functional enhancement are also critical. They become latent bugs in the future. The investigation found that totally 117 enhancement patterns appeared in multiple revisions. The presence of bug and enhancement modification patterns has seriously negative impacts on software development.

Also, 61 modification patterns were classified into refactoring. Those patterns do not directly have influences on development. However, in the viewpoint of lack of consistencies, they are unwelcome.

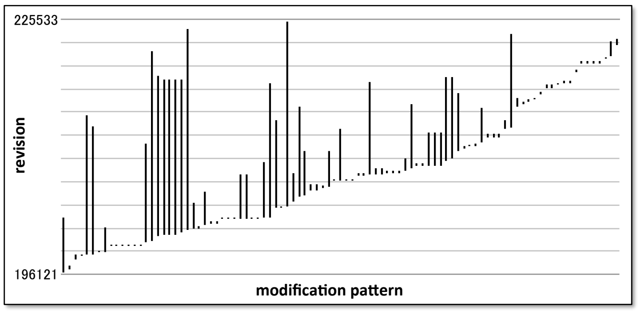
[1]In this experiment, we checked commit logs in the repository, surrounding code of modification patterns, and modifications after the target period.

Table II
CATEGORIES OF MODIFICATION PATTERNS IN RQ1

Software	Bug	Refactoring	Enhancement	Comment	Total
HTTPD	19	22	93	11	145
FreeBSD	23	39	24	8	94



(a) HTTPD



(b) FreeBSD

Figure 3. Appearance period of modification patterns where inconsistences arised provisionally (Investigation for RQ1)

We got 19 modification patterns related to comments. All of them are about changing western calendar in copyright statements. The presence of them does not affect the behavior of the system. However, from the viewpoint of software license, they should be modified consistently.

Figure 3 shows the appearance span of the 145 and 94 modification patterns. In X-axis, the patterns are arranged in the ascending order of the first appearance revision. In Y-axis, there is a line segment for every pattern. A line segment starts at the first appearance revision of a given pattern, and it ends at the last appearance revision. For example, in the case of FreeBSD, the leftmost modification pattern appeared between revisions 196,386 and 203,049 (see Figure 3(b)). The figure shows that most of the target revisions include at least one inconsistency. The proportions of including consistencies are 92.7% for HTTPD and 98.4% for FreeBSD.

Figure 4 shows the longest-period modification patterns found in HTTPD and FreeBSD, and Table III shows lists of revisions that those patterns appeared. Both the patterns are very simple. In the pattern of HTTPD, the conditional predicate of an if-statement was modified. In FreeBSD, a

```
     writev(c->fd,out, outcnt);
  #else
     write(c->fd,request,reqlen);
-    if (posting) {
+    if (posting>0) {
         write(c->fd,postdata,postlen);
         totalposted += (reqlen + postlen);
     }
```

(a) HTTPD

```
    if (ste_newbuf(sc, cur_rx) != 0) {
-       ifp->if_ierrors++;
+       ifp->if_iqdrops++;
        cur_rx->ste_ptr->ste_status = 0;
        continue;
    }
```

(b) FreeBSD

Figure 4. Modification patterns that appears over the longest period (In HTTPD, the modification pattern appears in revisions 83,946 and 88,628. In FreeBSD, the modification pattern appears in revisions 200,965, 213,438, 218,832, 223,648, and 223,951)

variable to be increased was modified. In both the software, the last revision that the patterns appeared is about 1 year and a half later than the first revision. Both the patterns deem to be bug fixes. That is, latent bugs of the overlooked code fragments lasted 1 year and a half longer than the same bugs in the other places.

As a result of this investigation, we answer RQ1 as follows: *About 19.7% and 9.6% of the detected modification patterns are ones appearing in multiple revisions in the target systems; About 92.7% and 98.4% of the target revisions included at least one of the inconsistencies that we detected.*

B. Investigation for RQ2

In order to answer RQ2, we identified overlooked code fragments from the entire source code of revisions 90,607 of HTTPD and 225,533 of FreeBSD by using the 737 and 983 modification patterns. The numbers of code fragments

Table III
LIST OF REVISIONS WHERE THE LONGEST-PERIOD MODIFICATION PATTERN OCCURRED

(a) HTTPD

Revision	Date	Modified file
83,946	1999-10-08 06:53:46	support/ab.c (2 places)
88,628	2001-04-02 15:19:45	support/ab.c

(b) FreeBSD

Revision	Date	Modified file
200,965	2009-12-25 05:43:31	sys/dev/ste/if_ste.c
213,438	2010-10-05 08:25:38	sys/dev/usb/net/usb_ethernet.c
218,832	2011-02-19 11:47:10	sys/dev/dc/if_dc.c
223,648	2011-06-29 01:16:43	sys/dev/gem/if_gem.c
223,951	2011-07-12 22:22:17	sys/dev/cas/if_cas.c (2 places)

Table V
EXECUTION TIME OF THE MINING AND DETECTION PROCESSINGS

Software	Mining processing	Detection processing
HTTPD	4 min. 58 sec.	12 sec.
FreeBSD	12 min. 16 sec.	7 min. 57 sec.

```
-    dump_avail[1] = phys_avail[1] - phys_avail[0];
+    dump_avail[1] = phys_avail[1];
```

(a) Modification pattern

```
phys_avail[1] = ctob(realmem);

dump_avail[0] = phys_avail[0];
dump_avail[1] = phys_avail[1] - phys_avail[0];

physmem = realmem;
```

(b) Detected bug(sys/mips/atheros/ar71xx_machdep.c)

Figure 5. An example of bugs detected in revision 225,533 of FreeBSD

```
-    ap.offset = (uap->offsetlo | ((off_t)uap->offsethi << 32));
+    ap.offset = PAIR32TO64(off_t,uap->offset);
```

(a) Modification pattern

```
ap.fd = uap->fd;
ap.offset = (uap->offsetlo | ((off_t)uap->offsethi << 32));
ap.len = (uap->lenlo | ((off_t)uap->lenhi << 32));
return (posix_fallocate(td, &ap));
```

(b) Detected refactoring candidate(sys/compat/freebsd32/freebsd32_misc.c)

Figure 6. An example of refactoring candidates detected in revision 225,533 of FreeBSD

that the tool pointed out were 18 for HTTPD and 94 for FreeBSD, each of which was carefully checked whether it was overlooked or not. The identification time were 12 seconds for HTTPD and 7 minutes 57 seconds for FreeBSD. Table V summarizes the execution time.

Table IV shows the investigation result. We found 15 bugs. Figure 5 shows an example of the bugs in FreeBSD. In this pattern, the way of calculating an array element, dump_avail[1] was modified. In the log of commit where this pattern appears, there is a description "*dump_avail layout should be sequence of [start, end) pairs, not <start, size>.*", which means that this pattern is a bug fixing. In revision 225,533, file ar71xx_machdep.c has the same code fragment as the patterns and their surrounding code are similar to each other, so that we counted this code fragment as "Bug".

We found 55 refactoring candidates. Figure 6 shows an instance in FreeBSD. In this pattern, an expression for shift and logical operations are replaced with a macro. This macro was added in revision 205,014. It was defined at file freebsd_misc.c. Hence, it is very easy to apply the suggested replacement. We considered that the suggested code fragment is an overlooked one of the refactorings performed in revision 205,014.

We found that 7 code fragments must be modified for consistent functional enhancements. Figure 7 shows an instance in FreeBSD. Before modification, the code fragment outputs the status code when SATA connect timeout happen meanwhile it also outputs actual time for the timeout after modification. At the detected code fragment (Figure 7(b)), variable "*timeout*" is available, so that it is very easy to apply the suggested enhancement to the detected code fragment.

Table IV
BREAKDOWN OF OVERLOOKED CODE FRAGMENTS THAT WERE POINTED OUT BY THE TOOL (INVESTIGATION FOR RQ2)

Software	Total	True positives				False positives	Precision
		Bug	Refactoring	Enhancement	Comment		
HTTPD	18	2	4	10	0	2	88.9%
FreeBSD	94	13	51	2	3	25	73.4%

```
-   device_printf(ch->dev, "SATA connect timeout status=%08x\n",
-   status);
+   device_printf(ch->dev,
+   "SATA connect timeout time=%dus status=%08x\n",
+   timeout * 100, status);
```

(a) Modification pattern

```
if (port < 0) {
  device_printf(ch->dev, "SATA connect timeout status=%08x\n",
  status);
} else {
```

(b) Detected code fragment(sys/dev/ata/ata-sata.c)

Figure 7. An example of code fragments overlooked in functional enhancement in revision 225,533 of FreeBSD

```
-  * Copyright (c) 2005-2010 Pawel Jakub Dawidek <pjd@FreeBSD.org>
+  * Copyright (c) 2005-2011 Pawel Jakub Dawidek <pawel@dawidek.ne
```

(a) Modification pattern

```
/*-
 * Copyright (c) 2005-2010 Pawel Jakub Dawidek <pjd@FreeBSD.org>
 * All rights reserved.
 *
```

(b) Detected code fragment(sys/geom/eli/g_eli_crypto.c)

Figure 8. An example of code fragments overlooked in comment in revision 225,533 of FreeBSD

There were 3 code fragments requiring modifications in comments because of copyright issues. Figure 8 shows an instance that we found. The western calendar and email address were forgotten to be modified.

Also, we found 2 and 25 false positives from the two target systems, respectively. They were certainly inconsistencies in the source code. They were regarded not as unintentional ones but as intentional ones. However, the numbers of the false positives are relatively small, so that the presence of the false positives should not be so impeditive to apply the proposed method to actual software maintenance.

As a result of this investigation, we answer RQ2 as follows: *We detected 16 and 69 unintended inconsistencies from a version of HTTPD and FreeBSD. In other words, we found an unintended inconsistency from every 8,229 and 51,739 lines of code in the versions. The precisions were 88.9% and 73.4%, respectively.*

C. Comparing with Clone-based Approaches

Clone-based approaches are used for detecting inconsistencies in source code [3]. As described in Section II, the purpose of the proposed method is detecting unintended inconsistencies that code clone detection tools cannot detect. In order to check that, we investigated whether the unintended inconsistencies of RQ2 were detected by code clone detection tools or not.

Herein, we used CCFinder[1] and Nicad[15] instead of CP-Miner [3] because they are widely used, open to the public, and easy to use. The characteristics of the tools are as follows:

- **CCFinder:** A token-based detection tool developed more than a decade ago. In the detection process, CCFinder replaces user-defined identifiers such as variable names or function names with special tokens, so that it can detect token-level inconsistencies.
- **Nicad:** Detecting code clones on block-level or function-level. Nicad identifies duplicated token sequences from every pair of blocks or functions by using LCS (Longest Common Subsequence) algorithm. If the ratio of detected duplication of a pair is larger than the threshold specified by the user, the pair is regarded as a code clone pair. Nicad can detect not only token-level inconsistencies but also statement-level ones.

In this experiment, we used the default settings of the tools. Table VI shows the detection results. CCFinder and Nicad did not detect the most part of the unintended incon-

Table VI
THE NUMBER OF RQ2 UNINTENDED INCONSISTENCIES THAT CCFINDER OR NICAD DETECTED

(a)HTTPD

Approach	Total	Bug	Refactoring	Enhancement	Comment
Ours	16	2	4	10	0
CCFinder	2	0	0	2	0
Nicad	2	0	0	2	0

(b)FreeBSD

Approach	Total	Bug	Refactoring	Enhancement	Comment
Ours	69	13	51	2	3
CCFinder	39	1	38	0	0
Nicad	2	1	1	0	0

Table VII
NUMERICAL DATA OF CCFINDER AND NICAD DETECTIONS

(a)HTTPD

Tool	# of clone classes	# of code clones	Execution time
CCFinder	1,004	3,435	6 sec.
Nicad	117	324	3 sec.

(b)FreeBSD

Tool	# of clone classes	# of code clones	Execution time
CCFinder	47,016	357,353	7 min. 35 sec.
Nicad	3,871	138,233	22 min. 26 sec.

sistencies that the proposed method detected. Why they were not detected by the tools is that they were not duplicated code chunk that the tool can detect as code clones. In the case of FreeBSD, CCFinder detected 38 inconsistencies for refactoring. All of them were changes of function names, which are token-level inconsistencies. They were not block-level duplication, so that Nicad did not detect them.

Table VII shows the numbers of clone classes and code clones detected by the tools. Both the tools detected many code clones, however, the detection results include only a small part of the unintended inconsistencies that the proposed method detected.

VII. Discussion

A. Normalization

The current tool performs only a simple normalization, which removes white spaces, tabs, and new-line characters. For most programming languages, white spaces and tabs are used only for layout of the source code. That is, removing them has no impacts on semantics of code fragments. However, some programming languages such as Python use them as structural information. If a target system is written with such a programming language, removing them is not an appropriate operation.

With only these simple normalizations, we cannot obtain all the overlooked code fragments. Assuming that there is a couple of code fragments that have the same operations and they are modified in the same way. If variable names are different, they are treated as different modification patterns in the mining process. The same problem also happens in the detection process.

If the tool performs more intelligent normalizations such as replacing variable names and literals with their types, we will obtain more inconsistencies. However, such normalizations require analyzing the entire source files. Currently, we analyze only code fragments appearing in 'svn diff' command, and so the amount of code to be analyzed is very small comparing to the entire of source code. Consequently, the tool has a high scalability. If we perform such intelligent normalizations by scanning all the source code of all the revisions, the scalability is drastically decreased.

B. No consideration of "added" code fragments

In the detection process, overlooked code fragments are searched based on text matching using code fragments before modification. Hence, in this approach, we cannot use modification patterns that represent code additions. In order to check how many such addition patterns exist, we extracted the patterns satisfying the following conditions.

- code fragment before modification is empty,
- there are 3 or more modification patterns whose code fragments after modification are the same.

As a result, we obtained eight patterns from the two programs. Four out of them are preprocessor instructions such as `#ifdef` and `#endif`. The other four patterns are the followings:

- addition of variable declaration,
- addition of label, which is for goto-statement,
- addition of method call, which unlocks semaphore, and
- addition of a closed brace.

A part of them, especially the method call addition, should be used for detecting overlooked code fragments. The others are not so valuable. For example, if a closed brace is missing, the program is not compilable. We can find such an easy problem with a compiler.

In order to use addition patterns, using surrounding code is one way. In the mining processing, if we detect code addition, its pattern is created with its surrounding code. Including a previous line and a latter one may be reasonable.

C. Number of Detected Unintended Inconsistencies

In this experiment, we found out 16 and 69 unintended inconsistencies in a single revision of HTTPD and FreeBSD. In LOC-based consideration, every 8,229 and 51,739 lines of code has an unintended inconsistency. The values are very different and the number of the targets is only two, so that we cannot conclude a general result on frequency of unintended inconsistency occurrences from the experiment.

On the other hand, the experiment revealed the followings:

- more than 90% target revisions of both programs contain one or more unintended consistencies, and
- the detected unintended inconsistencies by the proposed method were not detected by the clone detectors.

The above findings indicate that new approaches for detecting unintended inconsistencies are required and the proposed method can be a useful approach for that.

D. Manual Checking in Evaluation

In this experiment, the authors checked the code fragments detected by the tool. However, we are not developers of the target system, and the result of the manual checking may contain mistakes. Consequently, we have to conduct more experiments that developers check outputs of the tool.

E. Thresholds used in this experiments

In this experiment, we used only one combination of the three parameters, *support*, *confidence*, and *place*. If we used looser thresholds such as *support* is 2, more inconsistencies would be detected. In this experiment, we checked all the detected inconsistencies one-by-one. We are not the developers of the target systems, so that checking a detected inconsistency took a very long time. The total time of the checking was approximately 20 hours. In some cases, we spent more than an hour to check each of them. Hence, the number of detected inconsistencies with the thresholds was acceptable for manual checking by non-developers. If the inconsistency checking was performed by developers, the required time would be much shorter.

F. Comparison Tools

In this paper, we used two code clone detection tools, CCFinder and Nicad, for comparison. We used the default settings of the tools for detection. If we had used other settings (e.g., minimal code clone length), their detection results would have been better. Also, in this experiment, we only checked whether the tools could detect inconsistencies that were detected by the proposed method. Authors think that the proposed method has a complementary relationship with code clone detection tools for detecting inconsistencies. Code clone detection tools can identify inconsistencies even if they do not have the same modifications in the past.

We did not use historical clone-based approaches such as Saha et al., and Göde and Koschke developed [16], [8]. If we use such approaches, we can obtain how code clones have evolved. That is, code clones including inconsistencies are identified easier. However, note that, even if we use historical clone-based approaches, inconsistencies not included in code clones as we showed in Figure 1 are not identified.

VIII. Conclustion

In this paper, we proposed a new method to automatically detect code fragments to be modified for bug fix or functional enhancement. The proposed method detects overlooked code fragments by mining how code was modified in the past. The proposed method is free from issues of existing methods: (1) requiring a certain size of code chunk to be detected; (2) only token-level inconsistencies are detected.

The proposed method has been implemented as a software tool. It was applied to large-scale open source software. As a result, we obtained many code fragments to be modified for fixing bugs, functional enhancement, refactoring. Also, this application showed that the tool is enough scalable to be applied to large scale software. In the future, we are going to conduct more experiments that involve developers of the target systems. Also, we will try more intelligent normalizations. Those requires much more time to finish analysis; meanwhile we'll obtain more suitable results for eliminating inconsistencies.

Acknowledgment

This study has been supported in part by Grants-in-Aid for Scientific Research (A) (21240002), Grant-in-Aid for Exploratory Research (23650014), and Grand-in-Aid for Young Scientists (A) (24680002) from the Japan Society for the Promotion of Science.

References

[1] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code," *IEEE TSE*, vol. 28, pp. 654–670, 2002.

[2] Y. Higo, Y. Ueda, S. Kusumoto, and K. Inoue, "Simultaneous Modification Support based on Code Clone Analysis," in *Proceedings of APSEC '07*, 2007, pp. 262–269.

[3] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code," *IEEE TSE*, vol. 32, pp. 176–192, 2006.

[4] L. Aversano, L. Cerulo, and M. Di Penta, "How clones are maintained: An empirical study," in *Proceedings of CSMR '07*, 2007, pp. 81–90.

[5] L. Barbour, F. Khomh, and Y. Zou, "Late propagation in software clones," in *Proceedings of ICSM '11*, 2011, pp. 273–282.

[6] S. Bazrafshan, R. Koschke, and N. Gode, "Approximate code search in program histories," in *Proceedings of WCRE '11*, 2011, pp. 109–118.

[7] W. I. Chang and E. L. Lawler, "Approximate string matching in sublinear expected time," in *Proceedings of SFCS '90*, 1990, pp. 116–124 vol.1.

[8] N. Göde and R. Koschke, "Frequency and risks of changes to clones," in *Proceedings of ICSE '11*, 2011, pp. 311–320.

[9] F. Rahman, C. Bird, and P. T. Devanbu, "Clones: What is that smell?" in *Proceedings of MSR '10*, 2010, pp. 72–81.

[10] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An empirical study on inconsistent changes to code clones at the release level," *Science of Computer Programming*, vol. 77, no. 6, pp. 760–776, 2010.

[11] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *Proceedings of ESEC/FSE-13*, 2005, pp. 187–196.

[12] E. Duala-Ekoko and M. P. Robillard, "Clone region descriptors: Representing and tracking duplication in source code," *ACM TOSEM*, vol. 20, pp. 3:1–3:31, 2010.

[13] Z. Li and Y. Zhou, "PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code," in *Proceedings of ESEC/FSE-13*, 2005, pp. 306–315.

[14] B. Livshits and T. Zimmermann, "Dynamine: finding common error patterns by mining software revision histories," in *Proceedings of ESEC/FSE-13*, 2005, pp. 296–305.

[15] C. K. Roy and J. R. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization," in *Proceedings of ICPC '08*, 2008, pp. 172–181.

[16] R. K. Saha, C. K. Roy, and K. A. Schneider, "An automatic framework for extracting and classifying near-miss clone genealogies," in *Proceedings of ICSM '11*, 2011, pp. 293–302.