Problematic Code Clones Identification using Multiple Detection Results

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

***Abstract*—Most code clones are generated by copy-and paste programming. Copy-and-paste programming shortens a time required for implementation because pasted code is a template of the required functionality. However, it sometimes brings on new bugs to the source code. After copy-and-paste, pasted code is somewhat changed fitting for the context of the region surrounding the pasted code. For example, some identifiers are replaced with other identifiers or a few statements are inserted, deleted, or changed. If such modifications are incorrectly performed, bugs occur in code clones. However, not all code clones are problematic, many code clones have decent reasons for their existence. Consequently, simple code clone detection is inefficient for identifying problematic code clones.**

Firstly, this paper proposes a classification scheme for dividing problematic code clones from non problematic ones. Secondly, it proposes a method for extracting specific code clones classified as problematic ones. Thirdly, it presents results of case studies conducted for evaluating the proposed method. The proposed method uses multiple code clone detection tools, and it doesn't directly analyze program source code. After multiple detections, simple operations are performed to extract code clones that are likely to be problematic. In the case studies conducted on an open source software system, the proposed method could actually identify 22 problematic code clones.

***Keywords*-code clone; fault detection;**

I. Introduction

No software has no duplicate code. Recently, studies on duplicate code have been active and it attracts much attention. In the research area of duplicate code, a region duplicated to another region is called a **code clone**.

Copy-and-paste programming is the leading cause why code clones are introduced to program source code. For example, Kim et al. reported that developers averagely perform copy-and-paste 16 times per hour [1]. Copy-and-paste is a powerful tool when implementing a new functionality because it can immediately create *a template* of the new functionality. Developers have only to modify *the template* fitting for the context of the pasted region [2], [3].

However, code clones often make negative impacts on software development and maintenance. One of them is increasing bug occurrences. For example, if an instance of copy-and-pasted code is changed for fixing bugs or adding new features, its correspondents have to be changed simultaneously. If the correspondents are not changed inadvertently, bugs are newly introduced to them.

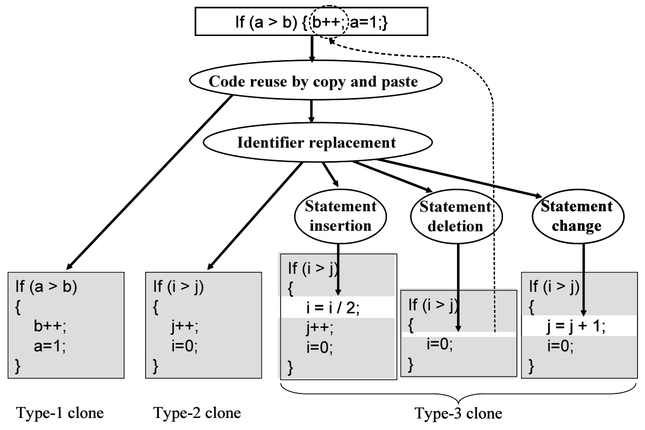
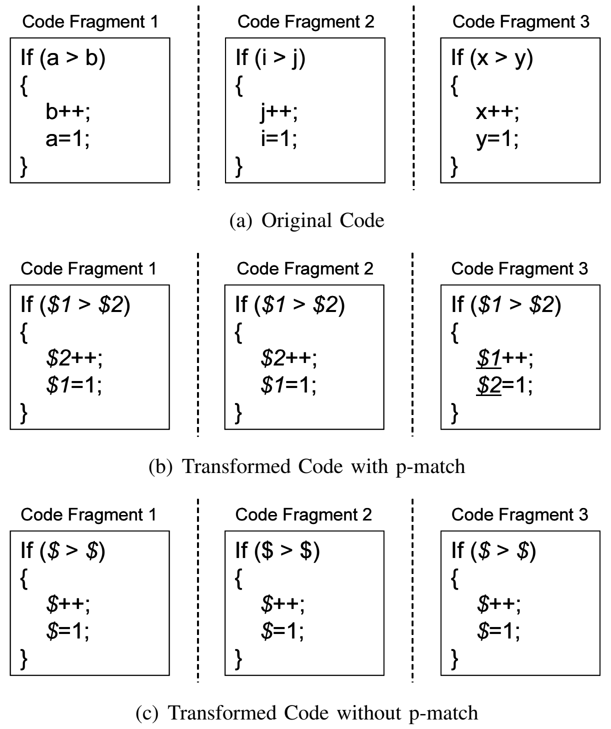


Figure 1. Code Reuse by Copy-and-Paste

Another example is small modification after copy-and-paste. Pasted code is often modified fitting for the context of the pasted region. As shown in Figure 1, identifiers such as used variables or invoked methods are changed, or some statements are added, deleted, or changed, or the both are performed. If such small modifications are incorrectly performed or forgotten to be performed, bugs are newly introduced into the pasted code. Here, we assume that a variable is forgotten to be replaced with another variable after paste: if the pasted code has different namespace from its original code, compiling the code is aborted and the mistake is identified; however, if the pasted code has the same namespace as the original code, compiling the code succeeds and the mistake isn't identified. It is difficult and costly to identify such a bug.

This paper proposes a new method for efficiently identifying bugs caused by copy-and-paste programming. The method utilizes the fact that *results of different detection tools are different from one another [4], [5].* At present, there are various code clone detection methods and implementations [6], and code clones are often operationally defined by individual detection methods. That is why some code clones detected by a certain detection method are not detected by another detection method. The proposed method detects code clones using multiple detection results, and



Code Fragment 1

```
If (a > b)
{
   b++;
   a=1;
}
```

Code Fragment 2

```
If (i > j)
{
   j++;
   i=1;
}
```

Code Fragment 3

```
If (x > y)
{
   x++;
   y=1;
}
```

(a) Original Code

Code Fragment 1

```
If ($1 > $2)
{
   $2++;
   $1=1;
}
```

Code Fragment 2

```
If ($1 > $2)
{
   $2++;
   $1=1;
}
```

Code Fragment 3

```
If ($1 > $2)
{
   $1++;
   $2=1;
}
```

(b) Transformed Code with p-match

Code Fragment 1

```
If ($ > $)
{
   $++;
   $=1;
}
```

Code Fragment 2

```
If ($ > $)
{
   $++;
   $=1;
}
```

Code Fragment 3

```
If ($ > $)
{
   $++;
   $=1;
}
```

(c) Transformed Code without p-match

Figure 2. Example of Code Transformation with and without Parameterized Match

analyzes their differences. The analysis identifies suspicious code clones. The analysis is fast and easy to implement, so that it can easily applied to practical size software systems.

II. Preliminaries

A. Existing Classification Scheme

Bellon et al. classified code clones based on how each code clone instance is different from its correspondents [4]. The classification is composed of three types (Figure 1 illustrates processes of each type code clone occurrence).

Type-1: Code clones that are identical to their correspondents. This type permits only the differences of white spaces and tabs.

Type-2: Code clones that include different identifiers from their correspondents.

Type-3: Code clones that include further differences than Type-2. Pasted code including statement insertion, deletion, or change is classified into this type.

B. Parameterized Match

Before introducing a new code clone classification scheme, *parameterized match* (*p-match*) must be explained. P-match is a code transformation rule for detecting more significant code clones. It transforms the same identifiers into the same special tokens before detection process. Figure 2 is an example of the code transformation. If a detection tool adopts p-match, the code is transformed as shown in Figure 2(b). In this example, code fragment 1 and code fragment 2 are detected as code clones. However, code fragment 3 is not regarded as a code clone of code fragments 1 and 2 because its special tokens don't correspond to the special tokens of the code fragments 1 and 2. If p-match is not used, the code is transformed as shown in Figure 2(c). In this example, each of the code fragments 1, 2, and 3 is detected as a code clone of the other code fragments.

P-match was initially proposed by Baker [7], and there are several detection tools that adopt p-match at present [8], [7]. P-match prevents from detecting code clones whose patterns of identifier use are different from their correspondents. In this paper, p-match is a key technique that divides problematic code clones from non problematic ones.

III. New Classification Scheme

The main purpose of a new classification scheme is dividing problematic code clones from non problematic ones. The new scheme is based on the Bellon's scheme described in Section II-A, and it is defined as follows.

Type-1a: Code clones that are exactly identical to their correspondents. Any kinds of textual differences are not permitted.

Type-1b: Code clones that include differences of white spaces and tabs. Any more differences are not permitted.

Type-2a: Code clones that include different identifiers. However, their patterns of identifier use are the same. This type code clones are detected by p-match detection tools.

Type-2b: Code clones that include different identifiers with different patterns of identifier use. This type code clones are not detected by p-match detection tools.

Type-3: Code clones that include statements insertion, deletion, or change.

In the new scheme, Bellon's type-1 is divided into type-1a and type-1b, and Bellon's type-2 is divided into type-2a and type-2b. Type-3 of the new scheme is the same as Bellon's type-3.

As described above, the new scheme's purpose is dividing problematic code clones from non problematic ones. We consider that type-1b, type-2b, type-3 of the new scheme are problematic code clones, and type-1a and type-2a are non problematic ones for the following reasons.

- The presence of type-1b code clones implies that different formats are used for implementing semantically same logics. Such format differences make code understanding more difficult.
- The presence of type-2b code clones implies that there may be incorrect replacements of identifiers after copy-and-paste or there may be code clones that were forgotten to be modified simultaneously when their correspondents were modified.

- As well as type-2b code clones, the presence of type-3 code clones implies that there may be incorrect statements insertion, deletion, or change after copy-and-paste, or there may be code clones that were forgotten to be modified simultaneously when their correspondents were modified.

IV. Proposed Method

The purpose of the proposed method is extracting only problematic code clones such as type-1b, type-2b, and type-3. Section IV-A explains the key idea of the proposed method, and Section IV-B to Section IV-F describe techniques for enhancing accuracy of the proposed method.

A. Key Idea

The key idea for extracting problematic code clones is very simple. We utilize the fact that *the detection result of a tool is different from the detection result of another tool*. At present, there are various code clone detection tools and each of them has a unique definition of code clones, so that the tools detect different regions of the same source code as code clones. Table I illustrates a model that different detection methods detect different types of code clones. For example, the method β detects only type-1a and type-1b code clones meanwhile the method ϵ detects all types of code clones. Here, we use this model for explaining our method.

In this explanation, we also assume that S^A_B is a set of type-B code clones detected by method A. Additionally, S^A_{all} is a set of all code clones that method A detects. Using this assumption, the following formula is formed:

$$S^A_{all} = S^A_{1a} \cup S^A_{1b} \cup S^A_{2a} \cup S^A_{2b} \cup S^A_3 .$$

If method A doesn't have a capability to detect type-B code clones, S^A_B becomes $\emptyset$.

Here, we focus on two detection method β and γ in Table I. Method β detects type-1a and type-1b code clones, and method γ detects type-1a, type-1b, and type-2a code clones, so that S^β_{all} and S^γ_{all} are represented as follows:

$$\begin{aligned} S^\beta_{all} &= S^\beta_{1a} \cup S^\beta_{1b} \\ S^\gamma_{all} &= S^\gamma_{1a} \cup S^\gamma_{1b} \cup S^\gamma_{2a} \end{aligned}$$

If S^β_{1a} and S^β_{1b} are the same as S^γ_{1a} and S^γ_{1b} respectively, S^γ_{2a} can be obtained by the following operation:

$$S^\gamma_{2a} = S^\gamma_{all} - S^\beta_{all}$$

Table I
Model of Detected Code Clone Types

Detection Method	Code Clone Type				
	1a	1b	2a	2b	3
method α	○	×	×	×	×
method β	○	○	×	×	×
method γ	○	○	○	×	×
method δ	○	○	○	○	×
method ϵ	○	○	○	○	○

```
/* Abuse this field as a pointer to the directory entry, used to
   find the expire list pointers */
dentry->d_time = (unsigned long) ent;

if (!dentry->d_inode)
{
   inode = autofs_iget(sb, ent->ino);
      if (IS_ERR(inode))
      {
         /* Failed, but leave pending for next time */
         return 1;
      }
   dentry->d_inode = inode;
}
```

(a) Before Normalization

```
dentry->d_time=(unsigned long)ent;
if(!dentry->d_inode){
inode=autofs_iget(sb,ent->ino);
if(IS_ERR(inode)){
return 1;}
dentry->d_inode=inode;}
```

(b) After Normalization

Figure 4. Example of Code Normalization

In a similar way, S^γ_{2a}, S^δ_{2b} and S^ϵ_3 can be obtained by the following operations respectively:

$$\begin{aligned} S^\delta_{2b} &= S^\delta_{all} - S^\gamma_{all} \\ S^\epsilon_3 &= S^\epsilon_{all} - S^\delta_{all} \end{aligned}$$

However, the above formulas are realized if the set of certain type code clones detected by a certain tool is equal to the set of the same type code clones detected by another tool. Unfortunately, this assumption is not realistic because each detection tool has its own definition of code clones, so that the same type code clones detected by different detection tools are always not exactly identical.

The remainder of this section describes countermeasures for the fact. Figure 3 depicts where each countermeasure is performed in the proposed method. In this paper, if code clones detected by different tools satisfy all the following conditions (Conditions 1 and 2), they are regarded as the same code clones and they are delisted from candidates of suspicious code.

B. Preparation 1: Code Normalization

Existing detection tools adopts various algorithms and heuristics for detecting more significant code clones [4]. For example, handling blank lines and comment lines are different between the tools. In order to reduce this problem, a code normalization is used. The code normalization consists of the following operations:

- blank lines and comment lines are removed;
- while spaces and tabs before/between/after lexical tokens are removed;

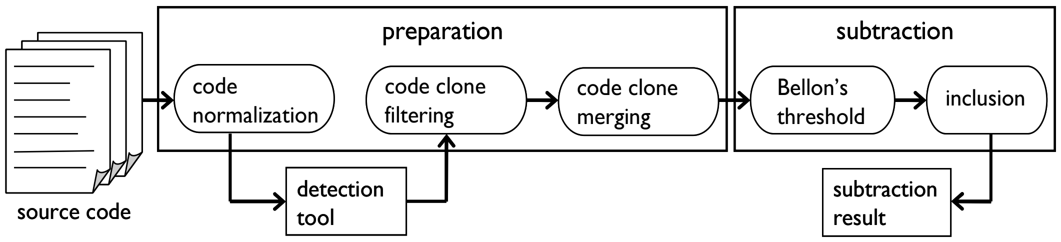


Figure 3. Overview of the Proposed Method

- lines including just a single open or close brace are removed, and the brace is joined to its above line.

Figure 4 represents an example of code before and after applying the code normalization. As shown in Figure 4, the code normalization reduces lines of code of source files. Therefore, if we use line-based detection techniques, we should carefully specify the length of code clones to be detected. We should specify a lower threshold than a value for the original source code. If we use other kinds of detection techniques, it is no problem that we use the same threshold for both original and normalized code. Other techniques such as token-based or AST-based are not affected by how each line consists of lexical tokens.

C. Preparation 2: Code Clone Filtering

Not all code clones detected by tools are beneficial for us. Code clone filtering introduced here is designed to remove unnecessary code clones and reduce false positives of subtraction result. We remove code clones in the same files, which overlap each other more than 30%. The condition is the same as the Kapser's study [9].

Code clones satisfying this condition are very like to be trivial ones such as consecutive variable declarations or consecutive case entries of switch statement, which are not worth to be investigated. We assume that a clone pair consists of two code clones, CF_1 and CF_2, $overlap(CF_1, CF_2)$ can be defined as follows:

$$overlap(CF_1, CF_2) = \frac{|\, lines(CF_1) \cap lines(CF_2)\, |}{|\, lines(CF_1) \cup lines(CF_2)\, |} \quad (1)$$

$lines(CF_1)$ and $lines(CF_2)$ are sets of lines included in code fragments CF_1 and CF_2 respectively.

D. Preparation 3: Code Clone Merging

If two clone pairs overlap each other, they are merged as a single clone pair. For example, we assume that there are two clone pairs, CP_1 and CP_2. Clone pair CP_1 (CP_2) consists of two code clones $CP_1.CF_1$ ($CP_2.CF_1$) and $CP_1.CF_2$ ($CP_2.CF_2$). If $CP_1.CF_1$ overlaps $CP_2.CF_1$ and $CP_1.CF_2$ overlaps $CP_2.CF_2$ respectively, the two code clones are removed from the set of detected clone pairs, and a clone pair CP_3 is added to it. In this case, CP_3 consists of two code clones, $CP_3.CF_1$ and $CP_3.CF_2$. Here the two code cones are as follows:

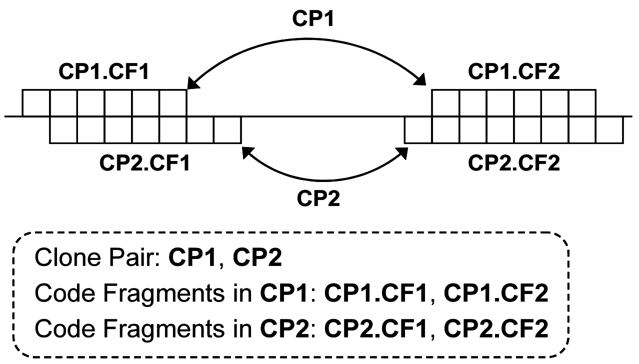

CP1
CP1.CF1
CP1.CF2
CP2.CF1
CP2.CF2
CP2
Clone Pair: CP1, CP2
Code Fragments in CP1: CP1.CF1, CP1.CF2
Code Fragments in CP2: CP2.CF1, CP2.CF2

Figure 5. An Example for Explaining Bellon's Threshold

$$\begin{aligned} CP_3.CF_1 &= lines(CP_1.CF_1) \cup lines(CP_2.CF_1) \\ CP_3.CF_2 &= lines(CP_1.CF_2) \cup lines(CP_2.CF_2) \end{aligned}$$

E. Condition 1: Bellon's Threshold

Bellon et al. proposed a threshold for determining whether code clones detected by different tools are the same or not [4]. Our proposed method also uses the Bellon's threshold. Bellon's threshold represents how a pair of code clones detected by a certain tool overlaps with a pair of code clones detected by another tool. Using formula 1, Bellon's threshold, *good value*, is defined as follows:

$$\begin{aligned} good(CP_1, CP_2) &= min(overlap(CP_1.CF_1, CP_2.CF_1), \\ &\quad overlap(CP_1.CF_2, CP_2.CF_2)) \end{aligned}$$

Range of good value is from 0 to 1. If good value is equal to or greater than threshold p, the two pairs of code clones are regarded as the same. Bellon et al. said that 0.7 is an appropriate value for good value [4]. In this paper, we use the same value as threshold p.

Figure 5 depicts an example for good value calculation. In this figure, there are two pairs of code clones, CP_1 and CP_2, and 4 code fragments (code clones), $CP_1.CF_1$, $CP_1.CF_2$,

$CP_2.CF_1$, and $CP_2.CF_2$. Code fragment $CP_1.CF_1$ overlaps with $CP_2.CF_1$, and $CP_1.CF_2$ overlaps with $CP_2.CF_2$ respectively. In this case good values becomes:

$$good(CP_1, CP_2) = min(\frac{5}{8}, \frac{6}{8}) = \frac{5}{8} \leq 0.7$$

In this case, the good value is less than 0.7, so that the two pairs of code clones are not regareded as the same.

F. Condition 2: Including

There is an important principle shared by all of the detection tools, that *the tools detect as much region as possible*. The fact implies that, the more different code the tool detects as code clones, the greater regions are detected as code clones. Therefore, we introduce the following condition: if two different tools detect the same code clones, the code clones detected by high detection capability tool always include the ones detected by low capability tool.

V. Case Study

In order to examine whether the proposed method can efficiently identify problematic code clones, a case study was conducted.

A. Target

The target of this case study is Linux Kernel (version 2.6.6). Especially, we focused on a directory, *linux-2.6.6/arch*, for closely investigating the result. The directory consists of 2,699 .c source files, and 769,467 lines of code. The reason of this choice is that some previous studies worked on this software [10], [11].

B. Configuration

This case study was intended to efficiently identify two kinds of bugs caused by copy-and-paste programming. They are identifier-level and statement-level incorrect modifications after copy-and-paste. We thought that identifier-level incorrect modifications generate type-2b code clones, and statement-level incorrect modifications generate type-3 code clones.

In order to efficiently obtain type-2b and type-3 code clones, we used two detection tools, CCFinderX [8] and Dude [12]. Table II presents the capabilities of the two detection tools.

Table II
Detection Capabilities of Dude, CCFinder, and CCFinderX

Detection Method	Code Clone Type				
	1a	1b	2a	2b	3
(a) CCFinderX with p-match	○	○	○	×	×
(b) CCFinderX without p-match	○	○	○	○	×
(c) Dude with gap=0	○	○	○	○	×
(d) Dude with gap=1	○	○	○	○	○

For extracting type-2b code clones, we subtracted the detection result of *(a) CCFinderX with p-match* from the detection result of *(b) CCFinderX without p-match*. In the remainder of this section, sets of code clones detected by *(a)* and *(b)* are represented by $S^{(a)}$ and $S^{(b)}$ respectively, and the difference set is represented by $S^{(b)-(a)}$.

For extracting type-3 code clones, we subtracted the detection result of *(c) Dude with gap=0* from the detection result of *(d) Dude with gap=1*. In the remainder of this section, sets of code clones detected by *(c)* and *(d)* are represented by $S^{(c)}$ and $S^{(d)}$ respectively, and the difference set is represented by $S^{(d)-(c)}$.

C. Result

For 4 sets of code clones, $S^{(b)}$, $S^{(b)-(a)}$, $S^{(d)}$, and $S^{(d)-(c)}$, we measured 2 values, *the number of the code clones* and *the total LOC of the code clones*. Table III summarizes the measurement results. In the case of type-2b, the number of code clones is dramatically decreased by the subtraction (6,980 → 524, decreasing 92%). However, in the case of type-3, the number of code clones is not so decreased by the subtraction (8,819 → 2,915, decreasing 70%).

We browsed the source code of all the code clones included in $S^{(b)-(a)}$ and $S^{(d)-(c)}$ to check whether each code clone falls under the following categories or not:

(I) *known bugs*, which have been already fixed in the later versions,
(II) *likely to be bugs*, which have not been fixed in the latest version yet,
(III) *other kinds of issues*, such as programming style.

Table III summarizes detection results, subtraction results, and the number of identified bugs. Note that we could not browse all the code clones included in $S^{(d)-(c)}$ because the number of them was very large. The subtraction result of (c) and (d) was far from our expectation. We had thought that the number of code clones in $S^{(d)-(c)}$ would be less than 1,000 as well as $S^{(b)-(a)}$.

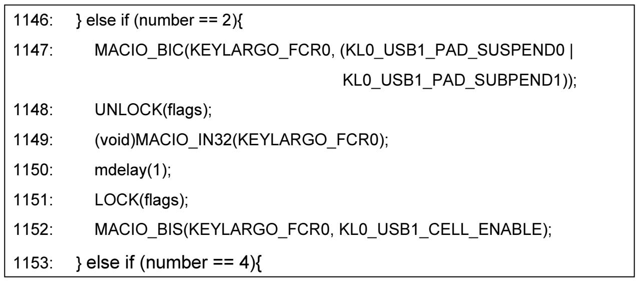
Table III
The Number of Clone Pairs

Type-2b Clone Identification

	$S^{(b)}$	$S^{(b)-(a)}$
LOCs of clone pairs	172,097	10,614
# of clone pairs	6,980	524
# of identified problems	-	11

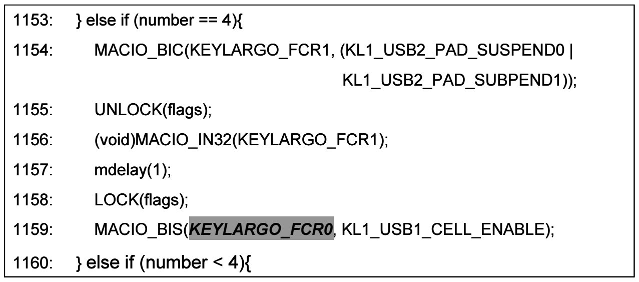
Type-3 Clone Identification

	$S^{(d)}$	$S^{(d)-(c)}$
LOCs of clone pairs	217,794	69,521
# of clone pairs	8,819	2,915
# of identified problems	-	11



```
1146:   } else if (number == 2){
1147:       MACIO_BIC(KEYLARGO_FCR0, (KL0_USB1_PAD_SUSPEND0 |
                                       KL0_USB1_PAD_SUBPEND1));
1148:       UNLOCK(flags);
1149:       (void)MACIO_IN32(KEYLARGO_FCR0);
1150:       mdelay(1);
1151:       LOCK(flags);
1152:       MACIO_BIS(KEYLARGO_FCR0, KL0_USB1_CELL_ENABLE);
1153:   } else if (number == 4){
```

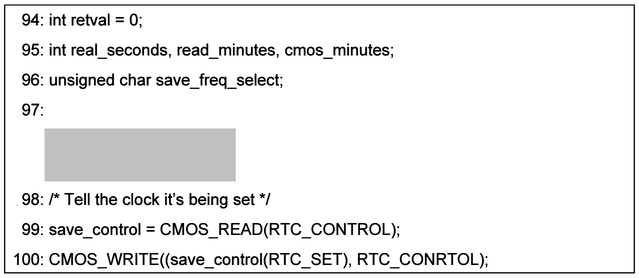
(a) Original Code (linux-2.6.6/arch/ppc/platforms/pmac_feature.c)



```
1153:   } else if (number == 4){
1154:       MACIO_BIC(KEYLARGO_FCR1, (KL1_USB2_PAD_SUSPEND0 |
                                       KL1_USB2_PAD_SUBPEND1));
1155:       UNLOCK(flags);
1156:       (void)MACIO_IN32(KEYLARGO_FCR1);
1157:       mdelay(1);
1158:       LOCK(flags);
1159:       MACIO_BIS(KEYLARGO_FCR0, KL1_USB1_CELL_ENABLE);
1160:   } else if (number < 4){
```

(b) Pasted Code (linux-2.6.6/arch/ppc/platforms/pmac_feature.c)

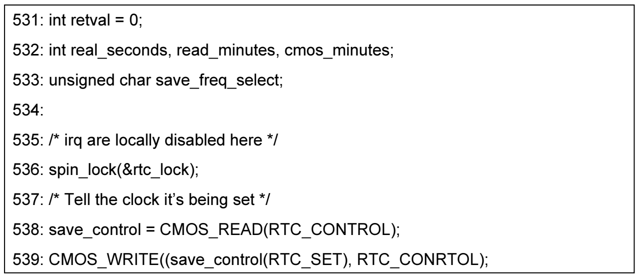
Figure 6. A Real Bug in Identified Type-2b Code Clones in the second case study



```
int retval = 0;
int real_seconds, read_minutes, cmos_minutes;
unsigned char save_freq_select;


/* Tell the clock it's being set */
save_control = CMOS_READ(RTC_CONTROL);
CMOS_WRITE((save_control(RTC_SET), RTC_CONRTOL);
```

(a) Buggy Code (linux-2.6.6/arch/mips/dec/time.c)



```
int retval = 0;
int real_seconds, read_minutes, cmos_minutes;
unsigned char save_freq_select;

/* irq are locally disabled here */
spin_lock(&rtc_lock);
/* Tell the clock it's being set */
save_control = CMOS_READ(RTC_CONTROL);
CMOS_WRITE((save_control(RTC_SET), RTC_CONRTOL);
```

(b) Its Correspondant (linux-2.6.6/arch/alpha/kernel/time.c)

Figure 7. A Real Bug in Identified Type-3 Code Clones in the first case study

1) Type-2b Code Clone Extraction: It took about 5 hours to check all the code clones included in $S^{(b)-(a)}$. In this investigation, we could identify a total of 11 problematic code clones. The number of identified problematic code clones was much less than the number that we had expected.

Figure 6 shows an identifier-level bug identified in $S^{(b)-(a)}$. The bug in Figure 6(a) is in the line 1159 of Figure 6(b). Variable KEYLARGO_FCR0 has to be replaced with KEYLARGO_FCR1.

2) Type-3 Code Clone Extraction: The large number of code clones in $S^{(d)-(c)}$ (2,915) prevented us from browsing all the code clones. We randomly chose some code clones from the set (the number of chosen code clones is 524, which is the same as the number of code clones in $S^{(b)-(a)}$). It took about 5 hours to check the chosen code clones. As a result, we could identify a total of 11 problematic clone pair.

Figure 7 shows a statement-level bug identified from the chosen code clones. The bug is that, there is no statement that calls spin_lock function in Figure 7(a). We couldn't identify which of Figure 7(a) and 7(b) is original code and which is pasted code. However, the code clones are very likely to have been generated by copy-and-paste, and our method could identify it as problematic one.

D. Discussion

We confirmed that subtraction results were much smaller than the original one. That means it is costless to use subtraction result for identifying problematic code clones. Besides, through this case study, in the case that we used the original results, we could not what kinds of problem were in the code clones, so that we had to very carefully investigate each code clone. On the other hand, in the case that used the subtraction result, we understand what kinds of problem were in the code clones. For example, if we investigate type-2b code clones, we concentrate on only consistency between code clones. Consequently, it is very likely that the proposed method shorten investigation time required for investigating a code clone.

VI. Related Work

There are some static analysis tools for finding latent bugs and issues that hinder software development and maintenance [13], [14]. They analyze program source code or Java byte code, and they identify problematic code. Their identifications are mainly based on heuristics, and various kinds of heuristics are adopted. However, such tools cannot identify bugs that were identified by the proposed method because the bugs can be identified only by detecting code clones. Therefore, the proposed method is not a rival of such static analysis tools. We can use the both of the proposed method and the static analysis tools at the same time for enhancing the quality of the program source code.

Li et al. developed a code clone detection tool, CP-Miner [11]. CP-Miner has a capability for detecting all types of code clones of the new classification described in Section III. CP-Miner also reports copy-and-paste related bugs using a metric *UnchangedRatio*. Our approach is different from the CP-Miner in the following points.

- CP-Miner identifies copy-and-paste related bugs from C/C++ source code. On the other hand, our approach is not restricted to a certain programming language because our approach doesn't directly analyze program source code. Our approach can be applied to source code of any program language if utilized detection tools can handle it.
- CP-Miner's bug identification is based on *UnchangedRatio*. The metric represents mapping relationship of only identifiers, and statements mapping is not counted. Hence, CP-Miner cannot detect copy-and-paste bugs on the statement-level. On the other hand, our approach can detect both identifier-level and statement-level bugs.
- CP-Miner uses a *UnchangedRatio* threshold for reducing false positive on bug identification. However, our approach doesn't use any filtering, so that there are so many false positives. In order to be sophisticated as more practical approach, some kinds of filtering false positives are mandatory.

Jiang et al. proposed a method for detecting bugs in code surrounding duplicated regions [10]. Inconsistencies of surrounding conditional statements such as if, while, and for are identified as bug candidates. For example, if the type of a conditional statement is different from the type of its corresponding conditional statement, they are bug candidates. Their approach can identify bugs not only in code clones but also in their surrounding code, which is a big advantage over other techniques for identifying copy-and-paste related bugs.

VII. Conclusion

In this paper, firstly we proposed a new classification scheme of code clones for dividing problematic code clones from non problematic ones. The new classification scheme is based on Bellon's classification, and it is simple and easy to understand. Secondly, we proposed a method for efficiently extracting a set of code clones falling into each category of the new classification. We can say that the proposed method can be applied to various kinds of software systems for the following reasons.

- The proposed method uses existing detection tools, and it doesn't directly analyze program source code. That means the proposed method can be applied to source code of any programming language if the used detection tools can handle the programming language. Currently, there are many detection tools and they can handle several popular programming languages such as Java, C/C++, C#, Visual Basic, and so on.
- The proposed method consists of simple operations, and it has high scalability.

Thirdly, the proposed method was applied to a famous real system, Linux Kernel. As a result of the application, we could identify *known bugs*, *likely to be bugs*, and *other kinds of issues* within a relatively short time frame.

Of course, there are so many things to do. As the next step of this research, we are going to do the followings.

- In the case study described in this paper, we didn't investigate false negatives, which are problematic code clones that were not extracted by the subtraction. However, for evaluating the usefulness of the proposed method more faithfully, we have to investigate false negatives.
- At present, we have used only three detection tools (CCFinder, CCFinderX, and Deckard). For evaluating the versatility of the proposed method, we have to conduct more case studies using other detection tools.

Acknowledgment

This work is being conducted as a part of Stage Project, the Development of Next Generation IT Infrastructure, supported by Ministry of Education, Culture, Sports, Science and Technology. It has been performed under Grant-in-Aid for Scientific Research (C)(20500033) supported by the Japan Society for the Promotion of Science.

References

[1] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An Ethnographic Study of Copy and Paste Programming Practices in OOPL," in *Proc. of 2004 International Symposium on Empirical Software Engineering*, Aug 2004, pp. 83–92.

[2] I. Baxter, A. Yahin, L. Moura, M. Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," in *Proc. of International Conference on Software Maintenance 98*, Mar 1998, pp. 368–377.

[3] M. Fowler, *Refactoring: improving the design of existing code*. Addison Wesley, 1999.

[4] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 804–818, Oct 2007.

[5] E. Burd and J. Bailey, "Evaluating Clone Detection Tools for Use during Preventative Maintenance," in *Proc. of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation*, Oct 2002, pp. 36–43.

[6] "Clone Detection Literature," http://www.cis.uab.edu/tairasr/clones/literature/.

[7] B. S. Baker, "Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1343–1362, Oct 1997.

[8] "CCFinderX," http://www.ccfinder.net/.

[9] C. Kapser and M. W. Godfrey, "Improved Tool Support for the Investication of Duplication in Software," in *Proc. of the 21st International Conference on Software Maintenance*, Sep 2005, pp. 305–314.

[10] L. Jiang, Z. Su, and E. Chiu, "Context-Based Detection of Clone-Related Bugs," in *Proc. of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, Sep 2007, pp. 55–64.

[11] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176–192, Mar 2006.

[12] R. Wettel and R. Marinescu, "Archeology of Code Duplication: Recovering Duplication Chanins From Small Duplication Fragments," in *Proc. of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, Sep 2005, pp. 63–70.

[13] "FindBugs," http://findbugs.sourceforge.net/.

[14] "PMD," http://pmd.sourceforge.net/.