Token Comparison Approach to Detect Code Clone-related Bugs

YongLee YII†, Yasuhiro HAYASE†, Makoto MATSUSHITA†, and Katsuro INOUE†

† Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama-cho, Toyonaka, Osaka 560-8531, Japan
E-mail: †{yii,y-hayase,matusita,inoue}@ist.osaka-u.ac.jp

Abstract Large software tends to have a significant amount of similar code, commonly known as code clones. Often the code clones are introduced through copy-and-paste process for code reuse purpose where the pasted code usually will go through some modifications such as renaming of identifiers and changing of parameters. In this paper, we propose a method using token comparison approach to detect bugs caused by abovementioned modifications. Our method tokenizes detected clones and performs a comparison to find out inconsistencies between them. The inconsistencies are then ranked based on predefined metrics to produce a bug candidate list. We implemented our method and tested on open source software. Our tool has found real bugs from the test subject. We believe our tool is suitable to serve the purpose of detecting code clone-related bugs in large software.

Key words Software Maintenance, Inconsistent Change, Code Clone-related Bug

1. Introduction

Recent studies [8] [9] show that large software systems contain a significant amount of similar code, commonly known as code clones or simply clones (hereafter we use them interchangeably). A pair of similar code fragments is called a clone pair. An equivalence set of a clone relation is called a clone set.

In many cases, code clones are introduced to the program through copy-and-paste process. While this practice can greatly reduce programming effort by reusing code in fast and easy way, it is prone to create bugs.

During code cloning, developer often modifies the pasted code in order to implement the desired functionality, but sometimes bugs are introduced. One of such modifications is to rename all instances of an identifier in pasted code. When this modification is done manually, there is possibility that the renaming is not completely applied to all relevant instances by mistake, therefore introducing unintended inconsistency which is considered as bug to the system.

Figure 1 shows an example of bug caused by inconsistent renaming of identifier. Code fragments consist of line 1486-1503 and line 1563-1579 are detected as a pair of code clones. This clone pair could be the result of copy-and-paste process. All variables named rc in pasted code fragment (line 1563-1579) have to be changed to retval but the one in line 1575 was left out. In consequence, the system displays wrong error code. This bug cannot be detected by compiler since the variable rc in pasted code fragment is still within the valid scope, thus produce no warning or syntax error.

According to [2] [3], unintended inconsistencies similar to abovementioned example are actually happened in production systems. In fact, the inconsistency shown in Figure 1 is detected by our proposed method (which will be described in detail later in this paper) in Linux version 2.6.6. This inconsistency was rectified in later version of Linux; therefore we can confirm that it is a genuine bug.

Although manual inspection is an effective method to find out bugs in programs, it would be too time-consuming and impractical for large software system. Jablonski et al. [4] proposed a tool called CReN, which is implemented as a plug-in to the integrated development environment (IDE) to track copy-and-paste activities and to ensure consistent renaming of identifiers. However, CReN is only applicable when coding a new program or to add new code to an existing program.

In this paper, we propose a method to detect bugs caused by inconsistent change of identifiers that exist in production systems. Our method first finds out inconsistencies between a pair of code clones based on token comparison. The inconsistencies are then going through some metric calculations to produce a list of potential bugs. In order to evaluate the effectiveness of our method, we implemented a tool based on proposed method and ran it on well-known large open source software.

The rest of this paper is organized as follows. Section 2 describes the proposed method in detail, and Section 3 gives the implementation details of our method. Section 4 explains the experiments that we performed and their results.
The general goal of lexical analysis in our approach is to transform the code clones detected in clone detection phase into structure that afterward fed into the mapping analysis step. Code clones that exist in target software are divided into tokens according to lexical rules of the programming language and each of these code clones will form a token sequence. While lexical analyzer scans through the code clones, it tokenizes them and identifies tokens made up by identifier. Comments and white spaces are eliminated, giving a normalized token sequence.

In a clone pair, if one code fragment is an exact copy without modification of another, or only variable, type or function identifiers were changed, the resulting normalized token sequences will have the same number of tokens with identifier at the same position (i.e. same token index). Tokenized clone pairs that fulfill this criteria will be selected for further processing in next step.

### 2.2 Mapping Analysis

In this step, mapping of identifiers in each clone pair is carried out. The clone pairs are in the form of token sequences which are the output of lexical analysis step. Each identifier in a code fragment is mapped to the identifier at the same position in another code fragment within a pair of code clones.

If the instances of an identifier in one code fragment are renamed to more than one identifier names, or not all instances of an identifier is renamed, inconsistencies are said to be occurred. For each unique identifier in a code fragment, the mapping is performed and its result is stored.

Table 1 shows one of such mapping results using the example shown in Figure 2. All identifiers exist in code fragment...
1 (line 127-139) are renamed consistently or remained the
same in code fragment 2 (line 161-173) except v_count and
varse. There is one of the instances of v_count (underlined
in Figure 2) is left unchanged in fragment 2. On the other
hand, instances of varse (italicized in Figure 2) are changed
to 2 different identifier names, arrays and lists, while one is
left unchanged. Inconsistencies are said to be occurred in
this 2 cases.

2.3 Result Filtering
Inconsistencies detected from the identifier mapping re-
sult are not necessary the case of unintended inconsistency
resulted from developer’s careless mistakes. In other words,
these inconsistencies are not always bugs. Therefore, they
need to go through some filtering algorithm in order to pro-
duce a list of potential bugs with less false positives. In our
approach, we use 2 metrics to serve the filtering purpose.

The first metric is UnchangedRatio as proposed in [2]. It
is defined as

\[
\text{UnchangedRatio}(v) = \frac{\text{NumOfUnchangedID}(v)}{\text{TotalNumOfID}(v)}
\]

For identifier \(v\), \(\text{NumOfUnchangedID}(v)\) is the number of oc-
currences of \(v\) in code fragment \(f_2\) where its name is remain
unchanged as compared to identifier at the same position
in code fragment \(f_1\). \(\text{TotalNumOfID}\) is the total number of
occurrences of \(v\) in \(f_1, f_1\) and \(f_2\) form a clone pair.

When UnchangedRatio equals to 0, it means all instances
of an identifier are being renamed. In contrast, when Un-
changedRatio equals to 1, it means all instances of an identi-
fier are remained in the same name. It is believed that if an
identifier is renamed in most of its instances and only a few
of its instances are not renamed, there is high possibility that
developer forgets to change them. Therefore, the smaller the
value of UnchangedRatio (except zero), the higher the possi-
bility of the not renamed identifier to be a bug. Inconsisten-
cies with the value of UnchangedRatio above the threshold
will be filtered out from bug candidate list. Since the orig-
inal code fragment in a clone pair cannot be determined,
UnchangedRatio has to be calculated in both directions.

Referring to Table 1, 4 instances of v_count are changed to
a_count in Fragment 2 while one instance is remained in the
same name, result in UnchangedRatio(v_count) = 0.2. Sim-
ilarly, 3 instances of varse are changed to arrays and lists
while one instance is left unchanged, result in Unchange-
dRatio(varse) = 0.25.

We propose another metric called Conflict to complement
UnchangedRatio since UnchangedRatio does not provide any
information if the instances of an identifier is changed to mul-
tiple names. When the instances of an identifier is changed
to multiple names, Conflict is set to true. In Table 1, varse
is one of such examples. In this case, the possibility that the
instances of an identifier are changed to multiple names in-
tentionally to implement the functionality is high. We filter
out this kind of inconsistencies even though their Unchange-
dRatio are below threshold value.

Table 1  Identifier Mapping Result for Clone Pair in Figure 2

<table>
<thead>
<tr>
<th>Identifiers in Code Fragment 1 (line 127-139)</th>
<th>Identifiers in Code Fragment 2 (line 161-173)</th>
<th>Occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>indx</td>
<td>indx</td>
<td>8</td>
</tr>
<tr>
<td>malloc</td>
<td>malloc</td>
<td>2</td>
</tr>
<tr>
<td>o_count</td>
<td>o_count</td>
<td>2</td>
</tr>
<tr>
<td>o_name</td>
<td>o_name</td>
<td>1</td>
</tr>
<tr>
<td>o_var</td>
<td>o_ary</td>
<td>2</td>
</tr>
<tr>
<td>v_count</td>
<td>a_count</td>
<td>4</td>
</tr>
<tr>
<td>v_name</td>
<td>v_name</td>
<td>2</td>
</tr>
<tr>
<td>varse</td>
<td>arrays</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>lists</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>varse</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>STORE_INCR</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 3  Overview of Proposed Bug Detection Method
3. Implementation

We have implemented a tool based on the proposed method. The system takes raw source files as input and produces a bug candidate list displayed on a graphical user interface as output. This section gives the details related to our implementation.

3.1 Code Clone Detection and Filtering

Code clone detection is an important step since code clone is the input to our approach. Therefore, the effectiveness of clone detection affects the result. There are plenty of tools that available for clone detection [1] [8] [10]. Among them, we have chosen CCFinder to perform the clone detection task in our approach. CCFinder is a token-based clone detection tool which offer good scalability and execution time to cope with large software. This tool is very effective in finding exactly identical clones and clones with only modification in identifiers.

One of the factors that affects the output of CCFinder is the minimum length of code clone. The lower the number, the more code clones will be found. In our experiments, we set the minimum length of a code clone to 30 tokens. This number was used in numerous previous researches [11] [12] on CCFinder and gave positive results.

CCFinder identifies a large amount of code clones, which some of them are deemed insignificant to our bug detection method. Therefore, it is important to filter the raw output of CCFinder and select only good candidate clones as the input to our proposed method. We applied the following two techniques to filter code clones detected by CCFinder and conducted experiments (details in section 4.2) to evaluate their effectiveness.

(1) Removing Highly Repeated Code Sequence Clones

There is a large portion of code clones detected by CCFinder consists of consecutive variable declarations and consecutive method invocations. These kinds of clones are mainly due to the structure of programming language and many of them are stereotyped code which is very stable. We filter out these kinds of clones using a metric called RNR(S) that represent the ratio of non-repeated code sequence in clone set S.

Let clone set S consists of n fragments, \( f_1, f_2, \ldots, f_n \). \( \text{LOS}_{\text{whole}}(f_i) \) represents the Length Of whole Sequence of fragment \( f_i \), and \( \text{LOS}_{\text{repeated}}(f_i) \) represents the Length Of repeated Sequence of fragment \( f_i \). \( \text{RNR}(S) \) is defined as

\[
\text{RNR}(S) = 1 - \frac{\sum_{i=1}^{n} \text{LOS}_{\text{repeated}}(f_i)}{\sum_{i=1}^{n} \text{LOS}_{\text{whole}}(f_i)}.
\]

The lower the value of \( \text{RNR}(S) \), the more repeated code sequences in clone set S. In our experiment, we set the RNR value to 0.5, filtering out clone set with the value less than 0.5 from the input to our method. This is the value proposed in [5].

(2) Removing Overlapping Clones

Among the clones detected by CCFinder, we discovered a lot of overlapping clones where a portion of code fragments in a clone pair overlap each other. Overlapping clones do not represent the nature of copy-and-paste process and most probably created coincidently. Since our method is based on the assumption of copy-paste-modify mechanism, overlapping clones are deemed insignificant to our method and need to be filtered out.

3.2 Graphical User Interface

Since our proposed method will produce a list of potential bugs for inspection, it is important to provides necessary information presented in decent way to ease the developers carry out this job. We built a graphical user interface to browse bug candidate list. A screen shot of the interface is shown in Figure 4.

The top left frame of the interface is a bug candidate list which can be sorted by the value of UnchangedRatio. One can start the inspection by looking at the most suspicious case (i.e identifiers with small value of UnchangedRatio). When clicking on an item in bug candidate list, the mapping result of selected identifier will be displayed on the bottom left frame. At the same time, the 2 frames on the right will display the related source files. Code clones where the selected identifier resided in are highlighted in the source files.

4. Experiments

In this section, we will describe the details of experiments that we have carried out with the tool implemented based
on the proposed method. The main purpose of experiments is to evaluate the effectiveness of our tool to detect real bugs in software systems.

We have chosen Linux version 2.6.6, which consist of 6,491 files and 4,364,540 lines of code as our experiment subject. We chose Linux because it is a well-known large scale production system, therefore we can test the capability of our tool to cope with large software system.

For the experiment results, we concentrate the discussion on 2 of the largest modules of Linux, arch and drivers.

4.1 Experiment Parameters

There are several parameters that will affect the experiment results. The following describes each of them and the value we set in our experiment.

(1) Threshold Value for UnchangedRatio

As mentioned in Section 2.3, a non-zero or non-one value for UnchangedRatio indicates inconsistent change of identifiers. When using this metric to narrow down the bug candidates, we are based on the idea that if the developer intents to changed all instances of an identifier consistently to another name but left out some of them. As such, smaller value of UnchangedRatio (except zero) better reveal a potential bug. In our experiments, we set the value of UnchangedRatio to 0.4, same to the value used in [2].

(2) Conflict Setting

We can specify the degree we tolerate inconsistency by determine how many different names an identifier can be changed to. In our experiment, if an identifier is changed to 2 different names or above, Conflict is set to true and it will be filtered out.

4.2 Code Clones Filtering

Table 2 shows the difference of the number of code clones detected by CCFinder from our test subject, Linux 2.6.6 before and after applying RNR filtering method. On the other hand, Table 3 gives the number of clone pairs after we took out the overlapped clones. The difference becomes obvious after we applied the RNR filtering.

The filtering can be set to on or off in the experiment. We have conducted experiments both with and without the filtering and we found that the filtering techniques greatly reduce our investigation effort by reducing the number of bug candidates created by insignificant clones. Therefore, we applied these filtering techniques when running our experiment with Linux 2.6.6.

4.3 Bugs Detection

The number of bug candidates found in Linux 2.6.6 is shown in Table 4. We have detected 87 bug candidates in arch module and 120 in drivers module.

If a clone pair contains a bug candidate, we call it suspicious clone pair. Adding up the lines of code (LOC) of all suspicious clone pairs gives total LOC of suspicious clones. Sometimes a clone pair can contain more than 1 bug candidate. In that case, we only add once when calculating the total LOC of suspicious clones. In both arch and drivers modules, the total LOC of suspicious clones occupies less than 0.1% of the total LOC. This gives a rough figure on the total number of lines that we need to review. In reality, we might need to review more code in order to verify a bug candidate, but it serve as a good start especially to detect bugs in a millions-line-of-code software.

One of the major tasks in our result analysis is to verify the bug candidates detected by our tool. We have inspected the bug candidates in arch and drivers modules. Some of the bug candidates are in high possibility. We checked them against Linux change log and later version of Linux. As a result, we were able to verify some of them to be the genuine bugs. Table 5 gives some instances of verified bug.

5. Discussion and Related Work

5.1 Limitation of Tool

Our tool currently cannot handle gapped clone that is created when a copy-and-paste code fragment gone through modifications such as insertion and deletion of statements. Gapped clones should be inspected as well since they also have the possibility of containing inconsistent renaming bugs. The greatest barrier for our current implementation to handle gapped clones is on the mapping analysis. Our method still lack of comprehensive algorithm to correctly map the identifiers in gapped clone. This gives a space for our tool to be improved.

---

Table 2 Number of Code Clones in Linux 2.6.6

<table>
<thead>
<tr>
<th>Module</th>
<th># Without Filtering</th>
<th># Filtered with RNR</th>
</tr>
</thead>
<tbody>
<tr>
<td>linux-2.6.6/arch</td>
<td>102,539</td>
<td>17,085</td>
</tr>
<tr>
<td>linux-2.6.6/drivers</td>
<td>159,764</td>
<td>44,881</td>
</tr>
</tbody>
</table>

Table 4 Number of Bug Candidates in Linux 2.6.6

<table>
<thead>
<tr>
<th>Module</th>
<th># File</th>
<th>Total LOC</th>
<th># Bug Candidate</th>
<th>Total LOC Suspicious Clones</th>
</tr>
</thead>
<tbody>
<tr>
<td>linux-2.6.6/arch</td>
<td>2,355</td>
<td>724,858</td>
<td>87</td>
<td>1,615</td>
</tr>
<tr>
<td>linux-2.6.6/drivers</td>
<td>2,323</td>
<td>2,344,594</td>
<td>120</td>
<td>3,637</td>
</tr>
</tbody>
</table>

Table 5 Genuine Bugs Found in Linux 2.6.6

<table>
<thead>
<tr>
<th>File Path</th>
<th>File Line</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>../arch/m68k/mac/iop.c</td>
<td>264</td>
<td>IOP_NUM_SCC</td>
</tr>
<tr>
<td>../arch/sparc/prom/memory.c</td>
<td>159</td>
<td>prom_phys_total</td>
</tr>
<tr>
<td>../arch/sparc64/prom/memory.c</td>
<td>117</td>
<td>prom_phys_total</td>
</tr>
<tr>
<td>../drivers/pci/hotplug/shpchp_ctrl.c</td>
<td>1575</td>
<td>rc</td>
</tr>
</tbody>
</table>
5.2 Related Work

Li et al. [2] have developed a tool called CP-Miner to detect copy-and-paste related bugs. CP-Miner uses data mining techniques to identify code clones and the bug detection is implemented as part of the tool. Before passing to bug detection process, code clones detected gone through a series of pruning procedures. CP-Miner has the ability to handle clone that is not exactly the same.

Recently, Jiang et al. [3] developed a tool to detect bugs in code clones and their surrounding code, called context. The clone detection component of the tool is based on DECKARD [13], a clone detection tool developed by them. Bug caused by renaming inconsistency is called a type-3 inconsistency in their work. The tool counts the number of unique identifiers in each code fragment within a pair of clones. Different number of unique identifiers is deemed as likely to be a bug.

6. Conclusion

In this paper, we introduced a method to detect bugs caused by inconsistent change of identifiers. We have also conducted experiments using Linux kernel in order to evaluate the effectiveness of our proposed method. As a result, we were able to discover bugs on this system. Therefore, we believe that our tool is effective for detecting bugs in large software systems.

In future, we would like to conduct more experiments on production software and perform a comprehensive analysis on the results. Also, we would like to improve our tool to handle gapped clones.

Acknowledgments

This work has been conducted as a part of EASE Project, Comprehensive Development of e-Society Foundation Software Program, supported by Ministry of Education, Culture, Sports, Science and Technology of Japan. The work was also supported by Japan Society for the Promotion of Science under Grant-in-Aid for Scientific Research (A)(17200001).

References