Simultaneous Modification Support based on Code Clone Analysis

Yoshiki Higo[†] Yasushi Ueda‡ Shinji Kusumoto[†] Katsuro Inoue[†]

[†]Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
[‡]JAXA's Engineering Digital Innovation Center, Japan Aerospace Exploration Agency
2-1-1 Sengen, Tsukuba, Ibaraki 305-8505, Japan
{higo, kusumoto, inoue}@ist.osaka-u.ac.jp, ueda.yasushi@jaxa.jp

Abstract

Maintaining software systems becomes more difficult as their size and complexity increase. One factor that makes software maintenance more difficult is the presence of code clones. A code clone is defined as a code fragment occurring more than once in identical or similar form into a software system. For example, the presence of code clones is a big factor of overlooking some places that should be modified simultaneously. One technique that helps the number of code clones is Refactoring. There are several research efforts that provide support to refactor code clones, but unfortunately some code clones cannot or should not be refactored (ex. stereotyped process, absence of abstraction functionality, performance enhancement). In order to support maintaining the consistency among code clones, we propose a simultaneous modification support method. Given a software system, firstly, a maintainer identifies a code fragment that must be modified. Then, only the code clones between the identified code fragment and the source files of the software system are detected. We developed a simultaneous modification support tool, Libra*, and applied it to open source software systems. The results showed that* Libra *was a good searching tool as much as* grep*, which is a useful tool of UNIX.*

1 Introduction

Software maintenance is defined as *the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment* [17]. As the size and the complexity of software increase, maintenance tasks become more difficult and burdensome. Arthur states that only one-fourth or one-third of all life-cycle costs are attributed to software development [3]. Also, Yip et al. says that some 67% of life-cycle costs are expended in the operation-maintenance phase of the life cycle [31].

A code clone is defined as a code fragment occurring more than once in identical or similar form into a software system (see section 2.1). The presence of code clones is one factor making software maintenance more difficult [28]. For example, if a code fragment is modified, it has to be determined whether or not modify each of its code clones. Unfortunately, we often overlook some of the code clones that should be modified simultaneously.

One technique that helps reducing the number of code clones is Refactoring. *Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior* [14]. By making refactoring efforts on a set of code clones, they can be merged into a method [8, 6, 15], a component [19], or an aspect [9].

Refactoring shall be an activity of perfective maintenance. The ISO Standard on Software Maintenance classifies perfective and adaptive maintenance as enhancements, and it also classifies corrective and preventive maintenance as corrections [18]. Some researchers have reported that enhancements are account for 80% of the total maintenance cost [1, 29]. That lets us know the importance of refactoring efforts on code clones.

But refactoring efforts may not always be a good solution to the code clone problem. An empirical study of Kim et al. revealed two points [23]: first one is that some code clones are short-lived, and merging them wouldn't improve the maintainability; second one is that most of long-living code clones are not suited to be refactored because there is no abstraction function of the programming language. Also, Balazinska et al. reported that differences between code clones tend to hinder merging them [5], which indicates that it requires countermeasures to modifying all code clones sharing the identical or similar form without overlooking.

In this paper, we describe a simultaneous modification method for code clones: firstly, a maintainer identifies the code fragment that must be modified; secondly, only the code clones between the code fragment and the source files of the system are detected. Detecting only the code clones of the identified code fragment has two advantages: the detection phase is much faster than detecting all the code clones in the system, and the maintainer are not confused by the information on unconcern code clones. We developed a simultaneous modification support tool, Libra, and applied it to open source software systems. The results showed that Libra is a good searching tool as much as grep, which is an useful tool of UNIX.

The rest of this paper is organized as follows. Section 2 provides the definition of code clone, and the description of the code clone detection tool CCFinder. Section 3 and 4 describe our proposed method and its implementation respectively. In Section 5 we present case studies using open source software systems, and discuss the results. Section 6 describes related work, section 7 concludes our study.

2 Code Clones

2.1 Definition

A code clone can be defined as a code fragment occurring more than once (in identical or similar form) into a software system. However, there is no single and generic definition of code clone. So far, several methods for detecting code clones have been proposed [4, 8, 13, 24, 27], and all of them have different definitions. Following describes primary code clone detection techniques.

Line based : Each line of the source code is compared with other lines.

Token based : After the source code is divided into tokens, same token sequences which are longer than a certain length are detected as code clones.

AST based : After building AST (Abstract Syntax Tree) from the source code, subtrees having the same structure are detected as code clones.

PDG based : After building PDG (Program Dependency Graph) as a result of semantic analysis of the source code, isomorphic subgraphs are detected as code clones.

Metrics based : After measuring several metrics from a certain unit (e.g. function, method, class) of software system, units having identical or similar metrics values to each other are detected as code clones.

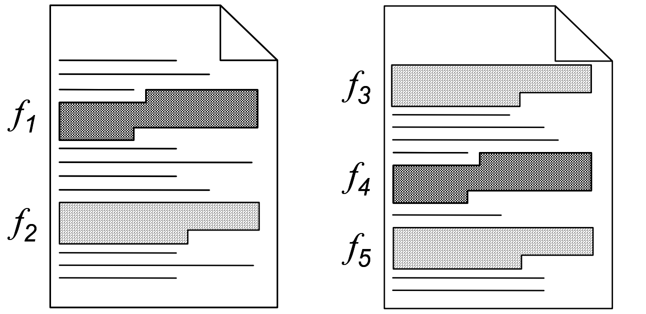


Figure 1. Clone Pair and Clone Set

In the past we have developed a code clone detection tool, CCFinder. CCFinder adopt token based technique in order to efficiently detect 'copy and paste' code clones. Burd et al. compared CCFinder and other code clone detection tools, and reported that CCFinder can detect much more code clones than the others [10]. Section 5.3 and 6 describe the bias derived from the use of CCFinder

2.2 CCFinder

In **CCFinder**[20], a clone relation is defined as an equivalence relation (i.e., a reflexive, transitive, and symmetric relation) on code fragments. A **code fragment** is a part of a source file and can be represented using a 5-tuple, (ID, $Line_{start}$, $Column_{start}$, $Line_{end}$, and $Column_{end}$). Given a code fragment f, $ID(f)$ is the unique numeric ID assigned by CCFinder to the file containing f. $Line_{start}(f)$ and $Line_{end}(f)$ are respectively the line numbers of the start and end of the code fragment. Similarly, $Column_{start}(f)$ and $Column_{end}(f)$ are respectively the column numbers of the start and end of the code fragment. With this definition, it is possible that some code fragments partially overlap. There is a clone relation between two code fragments if (and only if) the lexical token sequences are identical[1]. For a given clone relation, a pair of code fragments is called **clone pair** if the clone relation holds between them. An equivalence class of the clone relation is called **clone set**. That is, a clone set is a the largest possible set of code fragments, so that a clone relation exits between any pair of them.

Figure 1 illustrates an example of clone relation. In this figure, there are five code fragments, and a clone

[1] The sequences are the transformed as described below.

relation exists between the code fragments of the same color. Code fragment f_1 has a clone relation with code fragment f_4, and code fragments f_2, f_3, and f_5 have a clone relation with each other. In this example there are four clone pairs, (f_1, f_4), (f_2, f_3), (f_2, f_5), (f_3, f_5), and two clone sets, $\{f_1, f_4\}$, $\{f_2, f_3, f_5\}$.

CCFinder detects code clones from source files, and its output consists in the locations of the clone pairs. The minimum code clone length to be detected is set by the user in advance. The clone detection of CCFinder is a process consisting of the following steps.

1. **Lexical analysis**
   Each line of the source files is divided into lexical tokens corresponding to a lexical rule of the programming language. The lexical tokens of all source files are concatenated into a single token sequence.

2. **Normalization**
   The token sequence is transformed, i.e., tokens are added, removed, or changed based on the transformation rules that aim at regularization of identifiers and identification of structures.

3. **Match detection**
   From all the sub-strings on the transformed token sequence, identified pairs are detected as clone pairs.

4. **Formatting**
   Each location of the detected clone pairs is converted into the line and column numbers on the original source files.

3 Proposed Method

3.1 Motivation

The presence of code clones makes the maintenance process more complicated. For example, when a bug is found and a code fragment including the bug is modified, it is necessary to determine whether or not each code clone in the same clone set has to be modified. It is likely to overlook some of them. Thus, it is important to have a feature for searching the code clones of a specified code fragment.

We propose a modification support method for code clones. The method supports simultaneous modifications by showing the code clones of the code fragment that the user is going to modify. The proposed method can prevent the user from overlooking some code fragments of the source files in the modification process, and enable him/her to effectively maintain the system.

3.2 Approach

In this paper, a simultaneous modification support method is proposed. The method is used after a maintainer identified a code fragment that must be modified. The method detects only the code clones between the identified code fragment and the source files of the system.

To detect only the code clones between the identified code fragment and the source files, we use the following options of CCFinder.

-cg- : do not detect code clones across groups,

-cf- : do not detect code clones across files,

-cw- : do not detect code clones within a file,

A group is a set of source files. The user can freely construct groups before CCFinder's code clone detection. Usually, we build a group from source files which are in the same directory or in the same module. Building groups enables to measure the similarities at directory or module level.

In the proposed method, code clone detection is performed on the following steps.

1. **Group setting** : Assign the identified code fragment to group 1, and the source files to group 2.

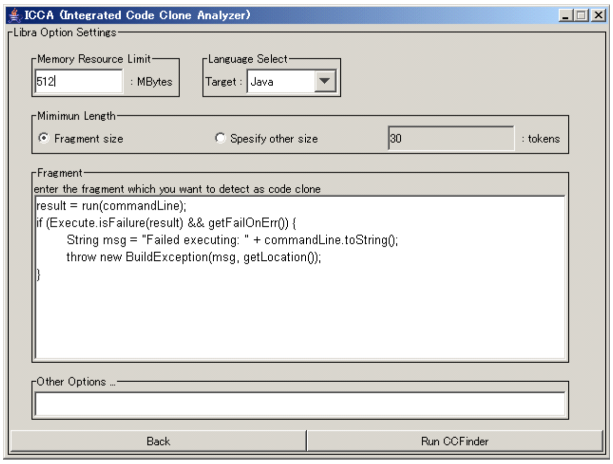
2. **Option setting** : Set options -cf- and -cw-.

3. **CCFinder execution** : Run CCFinder from the identified code fragment and the source files.

Code clone detection on the above steps allows us to get only the code fragments that are identical or similar to the identified code fragment in the source files. If we execute CCFinder without the options, CCFinder detects all code clones included in the source files.

Also, the proposed method counts the number of lexical tokens of the identified code fragment, and set it as minimum length of code clones which CCFinder detects. This prevents CCFinder from detecting redundant code clones.

Detecting only the minimum necessary code clones (in other words, only the code clones between the identified code fragment and the source files) has two advantages: the detection speed is much faster than detecting all code clones in the system, and maintainers are not confused by the information of unconcern code clones.



Memory Resource Limit: 512 : MBytes

Language Select: Target : Java

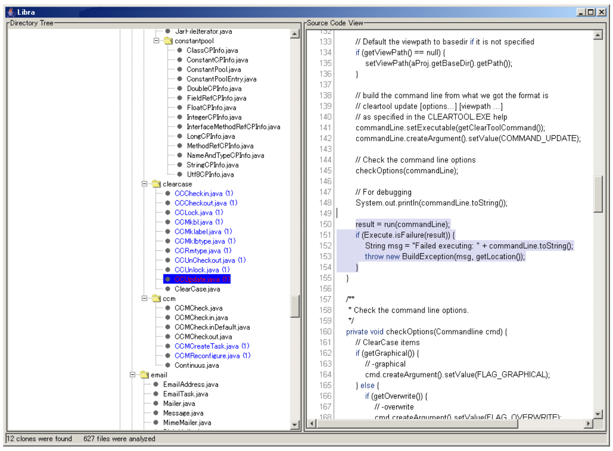
Mimimun Length: ● Fragment size ○ Specify other size 30 : tokens

Fragment: enter the fragment which you want to detect as code clone

```
result = run(commandLine);
if (Execute.isFailure(result) && getFailOnErr()) {
        String msg = "Failed executing: " + commandLine.toString();
        throw new BuildException(msg, getLocation());
}
```

Other Options ...

(a) Input fragment



```
JarFileIterator.java
constantpool
    ClassCPInfo.java
    ConstantCPInfo.java
    ConstantPool.java
    ConstantPoolEntry.java
    DoubleCPInfo.java
    FieldRefCPInfo.java
    FloatCPInfo.java
    IntegerCPInfo.java
    InterfaceMethodRefCPInfo.java
    LongCPInfo.java
    MethodRefCPInfo.java
    NameAndTypeCPInfo.java
    StringCPInfo.java
    Utf8CPInfo.java
clearcase
    CCCheckin.java (1)
    CCCheckout.java (1)
    CCLock.java (1)
    CCMkbl.java (1)
    CCMklabel.java (1)
    CCMklbtype.java (1)
    CCRmtype.java (1)
    CCUnCheckout.java (1)
    CCUnlock.java (1)
    CCUpdate.java (1)
    ClearCase.java
ccm
    CCMCheck.java
    CCMCheckin.java
    CCMCheckinDefault.java
    CCMCheckout.java
    CCMCreateTask.java (1)
    CCMReconfigure.java (1)
    Continuus.java
email
    EmailAddress.java
    EmailTask.java
    Mailer.java
    Message.java
    MimeMailer.java
```

```
        // Default the viewpath to basedir if it is not specified
        if (getViewPath() == null) {
            setViewPath(aProj.getBaseDir().getPath());
        }

        // build the command line from what we got the format is
        // cleartool update [options...] [viewpath ...]
        // as specified in the CLEARTOOL.EXE help
        commandLine.setExecutable(getClearToolCommand());
        commandLine.createArgument().setValue(COMMAND_UPDATE);

        // Check the command line options
        checkOptions(commandLine);

        // For debugging
        System.out.println(commandLine.toString());

        result = run(commandLine);
        if (Execute.isFailure(result)) {
            String msg = "Failed executing: " + commandLine.toString();
            throw new BuildException(msg, getLocation());
        }
    }

    /**
     * Check the command line options.
     */
    private void checkOptions(Commandline cmd) {
        // ClearCase items
        if (getGraphical()) {
            // -graphical
            cmd.createArgument().setValue(FLAG_GRAPHICAL);
        } else {
            if (getOverwrite()) {
                // -overwrite
```

12 clones were found   627 files were analyzed

(b) Detection result

Figure 2. Snapshots of Libra

4 Implementation

We implemented a tool, named **Libra**[2], based on the above-mentioned method. At first, the user inputs the code fragment to be modified and specifies the target source files. Figure 2(a) shows the window where the user inputs the code fragment. Next, Libra executes CCFinder with the options previously described. Figure 2(b) shows the window of the detection result. On the left side, the target files are listed as a directory tree. Each file containing code clones of the input code fragment is shown highlighted in color, with the number of contained code clones next to it. When the user selects a file in this view, the right side displays its source code.

```
  ir_debug( Dmsg(10, "ProcWideReq3 start!!\n") );

# buf += HEADER_SIZE;  Request.type3.context = S2TOS(buf);
# buf += SIZEOFINT;    Request.type3.buflen = S2TOS(buf);

  ir_debug( Dmsg(10,
                 "req->context = %d\n",
                 Recuest.type3.context) );
```

(a) Before modification (version 3.6)

```
  ir_debug( Dmsg(10, "ProcWideReq3 start!!\n") );

+ if (Request.type3.datalen != SIZEOFSHORT * 2)
+ return( -1 );
+
  buf += HEADER_SIZE;  Request.type3.context = S2TOS(buf);
  buf += SIZEOFINT;    Request.type3.buflen = S2TOS(buf);

  ir_debug( Dmsg(10,
                 "req->context = %d\n",
                 Recuest.type3.context) );
```

(b) After modification (version 3.6p1)

Figure 3. An example of Canna's modifications

5 Evaluation

We have applied Libra to actual modification instances on open source software. We chose Ant [2](version 1.6.0) and Canna [11](version 3.6) as our targets.

Table 1 illustrates some attributes of the target software. Since Libra is a tool for searching code fragments to be modified simultaneously in the maintenance process, we have evaluated whether Libra could find out such code fragments.

We compared Libra's search results to the search results obtained running grep, which is a command line search tool of UNIX. In this case study, we regarded the locations modified in the post-version as the correct set of code fragments that Libra and grep should identify.

5.1 Canna

Canna 3.6 includes 92 .c files and 40 .h files, and the size is 99,747 LOC. We used an actual modification occurred between version 3.6 and version 3.6p1. In version 3.6p1, a buffer overflow check is inserted in 21 places, before each process using buffers. Figure 3(a) represents a code fragment before being modified, and Figure 3(b) represents its modified version. The lines beginning with '+' were added in version 3.6p1.

We assumed that a maintainer had identified a code fragment of them, and input it into Libra in Figure 2(a). We investigated whether Libra could detect the 21 code fragments. Two lines beginning with '#' were the input

[2] http://sel.ist.osaka-u.ac.jp/icca/libra-e.html

Table 1. Attributes of target software

Name	Version		Total LOC		# Files		Language
	pre	post	pre	post	pre	post	
Canna	3.6	3.6p1	99,747	99,867	96	96	C
Ant	1.6.0	1.6.1	180,844	159,613	627	631	Java

of Libra, and 17 code fragments were detected as identical or similar to it. All the detected code fragments were the modified code fragments in version 3.6p1, but Libra couldn't detect 4 modified code fragments. The reason why they were not detected is that they did not have the two lines consecutively.

As the input of grep, we chose string Request.type which is part of the variable name used in the buffer process. The reason why we didn't use the whole of the variable name is that there are many variables having similar names to Request.type3.context. They were used in different functions, but the functions had similar logics. Request.type is the common character sequence of the variables located at the places modified in version 3.6p1. As result, grep detected 58 Request.types, and 20 modified code fragments were included[3].

5.2 Ant

Ant 1.6.0 includes 627 .java files, and the size is about 180,844 LOC. We used an actual modification between version 1.6.0 and version 1.6.1. This modification was the addition of log instructions in 10 places. Figure 4(a) represents a code fragment before being modified, and Figure 4(b) represents its modified version. The lines beginning with '+' were added and the line with '!' was changed.

We assumed that a maintainer had identified the code fragment of Figure 2(a), and input it into Libra. We investigated whether Libra could detect the 10 code fragments. The five lines beginning with '#' were the input of Libra, and 12 code fragments were detected as identical or similar to it. All 10 modified code fragments were included in them, and the other 2 code fragments were not modified in version 1.6.1. Because this modification is not a bug fix but an addition of logging instructions, we can't judge whether the 2 code fragments should be modified or not. However, pointing out such code fragments may be helpful to the maintainer in some situations.

As the input of grep, we chose the string Execute.isFailure which is part of the if-statement condition. As a result, grep detected 25 Execute.isFailures, and all modified code fragments were included[4].

```
  commandLine.setExecutable(getClearToolCommand());
  commandLine.createArgument().setValue(COMMAND_CHECKIN);

  checkOptions(commandLine);

# result = run(commandLine);
# if (Execute.isFailure(result)) {
#   String msg = "Failed executing: " + commandLine.toString();
#   throw new BuildException(msg, getLocation());
# }
```

(a) Before modification (version 1.6.0)

```
  commandLine.setExecutable(getClearToolCommand());
  commandLine.createArgument().setValue(COMMAND_CHECKIN);

  checkOptions(commandLine);

+ if (!getFailOnErr()) {
+     getProject().log("Ignoring any errors that occur for: "
+     + getViewPathBasename(), Project.MSG_VERBOSE);
+ }

  result = run(commandLine);
! if (Execute.isFailure(result) && getFailOnErr()) {
    String msg = "Failed executing: " + commandLine.toString();
    throw new BuildException(msg, getLocation());
  }
```

(b) After modification (version 1.6.1)

Figure 4. An example of Ant's modifications

Table 3. Code clone detection result in the case that the threshold is 20 tokens (without the proposed method)

Target	# detected code clones	# run-time
Canna	8,747	27 secs.
Ant	31,372	53 secs.

5.3 Discussion

Table 2 illustrates precisions, recalls and execution times of Libra and grep. While the recall values between them are almost the same, the precision values

[3] grep returned 234 lines, and 134 lines were related with a modified code fragments.

[4] grep returned 25 lines, and 10 lines were related with a modified code fragments.

Table 2. Comparison between Libra and grep

Target	Libra			grep		
Name	recall	precision	run-time	recall	precision	run-time
Canna	81%	100%	8 secs.	95%	34%	less than 1 sec.
Ant	100%	83%	10 secs.	100%	40%	less than 1 sec.

of Libra are better than grep. This means that the results of Libra include less extra information. In other words, maintainers can avoid spending time to check code fragments which don't need to be modified.

We also compared the execution time of Libra and grep. The execution time of grep was less than a second in both cases while Libra took 8 and 10 seconds respectively. Libra's execution times were far larger than the execution times of grep, but we believe it is not a problem considering the high precision.

As a matter of course, it is possible to use other code clone detection techniques [4, 8, 13, 24, 27] for simultaneous modification. However we believe that CCFinder is better than other techniques/tools because of two reasons: first one is that CCFinder can handle multiple programming languages that are widely used; second one is that CCFinder can detect much more code clones than others from the same code base, which was empirically proved by Burd et al. [10]. In the context of simultaneous modification, it is very important to detect code fragments to be modified without overlooking.

The rest of this section describes the case that we detected all code clones in the source code. In that case, we used Gemini, which is an software tool for investigating code clones in a software system [16]. When using Gemini, we have to specify the minimum token length of code clones to be detected because we don't specify the target code fragment. In contrast, in the proposed method the appropriate threshold is automatically specified from the size of the input code fragment. It is difficult to manually specify the appropriate threshold. If we specified a too big value, CCFinder wouldn't detect code clones that we concern. If we specified a too small value, enormous number of code clones would be detected and we would be confused by the unconcern code clone information.

We specified 20 tokens as the threshold. Table 3 represents the detection results. An enormous number of code clones were detected, but most of them were not required to the modifications; in the case of Canna, we wanted the location information of only 21 code clones; in the case of Ant, we wanted 10 code clones.

The detection time is much slower than the proposed method. This is due to CCFinder detection process: CCFinder outputs code clones after sorting the detected clone pairs and generating clone sets. A large amount of unconcern code clones slowed the detection process.

From the comparison results with Gemini usage, we can say that the proposed method can quickly detect only the code clones that we need.

6 Related Work

Kim et al. performed experiments on the repositories of open source software systems to investigate how code clones appear and disappear [23]. The experimental results revealed the following points, which, in past, motivated us to propose a simultaneous modification support method.

- Some code clones are short-lived. Merging them wouldn't improve the maintainability.
- Most of long-living code clones are not suited to be refactored because there is no abstraction function of the programming language.

Kapser et al. also suggested that code clones are not always harmful from their experiences [21]. They reported several situations where code duplication is a reasonable or even beneficial way to handle large-scale complex software systems: for example, when developing a new driver for a certain hardware family, there may be already drivers to handle some other hardware families; however, there are often considerable differences in the functionalities or features between the families, so that it is very risky and unrealistic to merge code clones in the drivers of them. Nevertheless, if a bug was found in the driver of a certain hardware family, it would be very likely that there are the same bugs in the drivers of the hardware families having similar features to the family. We would have to find and correct each bugs in the drivers without overlooking. In cases like this, simultaneous modification can be a great support of software maintenance.

Balazinska et al. reported what kinds of differences between code fragments tend to hinder applying reengineering actions to them [5]. The result was quite natural, that strictly identical or superficially different code clones are easier to reengineer than code clones including other kinds of differences. In other words, code clones including some gaps are difficult to be refactored. However, CCFinder can detect only identical or renamed code clones[5], that is, it cannot detect gapped code clones. It may be more effective to use another clone detection technique that can detect gapped code clones in the context of simultaneous modification. Some of those techniques/tools are metrics-based detection [27] and CP-Miner [25].

Toomim et al. has proposed a synchronous modification method on code clones included in the same clone set [30]. In their method, there is a database of code clone information in the backend of the editor program. And when a code fragment included in a clone set is modified, other code fragments in the clone set are also simultaneously modified. Duala-Ekoko at al. also has proposed a simultaneous editing method [12]. The method identifies corresponding lines in code fragments being similar to each other based on the Levenshtein distance[6] after the code fragments are detected. A implementation of the method has been fully integrated in Eclipse. At the present, the method can handle only clone pairs, in other words, cannot handle clone sets consisting of there or more code fragments. Since the both methods have a critical drawback, they cannot be used in real software development or maintenance. The drawback is that they don't consider the differences between code fragments to be edited simultaneously despite there are often small differences between the code fragments. Their methods works well on only identical code clones, which don't include different identifiers, or reordered parameters.

Mann suggested that it should be effective to track 'copy and paste' actions [26]. That enables us to know where arbitrary code fragment were derived from, and to modify all of the code fragments derived from the same source simultaneously. 'Copy and paste' is one reason why there are code clones in the source code; for example, Kim et al. reported that developers do block or method level coy-and-paste actions approximately four times per hour [22]; also, Balint et al. reported that there often happens inconsistencies between 'copy and pasted' code [7]. Tracking 'copy and paste' should be able to identify many code clones. The advantage of this method is that we can identify any kinds of code clones as long as they are generated by 'copy and paste'. Each code clone detection technique depends on its detection algorithm, in other words, it cannot detect all kinds of code clones. Tracking 'copy and paste' should be a good support for simultaneous modification.

7 Conclusion

In this paper, we proposed a simultaneous modification support method for code clones. The proposed method detects only the code clones between the code fragment to be modified and the source files. In the process of source code modification (ex. debugging, adding new functions), this method is deemed to be effective.

We have also implemented a software tool, Libra based on the proposed method and applied it to open source software systems. The application results showed that the precisions of Libra is better than the precision of grep, but the detection with Libra is slower than with grep.

Acknowledgements

This work has been conducted as a part of EASE Project, Comprehensive Development of e-Society Foundation Software Program, and Grant-in-Aid for Exploratory Research(186500006), both supported by Ministry of Education, Culture, Sports, Science and Technology of Japan. Also it has been performed under Grant-in-Aid for Scientific Research (A)(17200001) supported by Japan Society for the Promotion of Science. We would like to express our thanks to Simone Livieri, a Ph.D student of our LAB for proofreading this paper.

References

[1] A. Abran and H. Nguyenkim. Analysis of maintenance work categories through measurement. In *Proc. of the Conference on Software Maintenance*, pages 104–113, Oct 1991.

[2] Ant. http://ant.apache.org/.

[3] L. Arthur. *Software Evolution: The Software Maintenance Challenge.* Wiley, 1988.

[4] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proc. of the 2nd Working Conference on Reverse Engineering*, pages 86–95, Jul 1995.

[5] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Proc. of the 6th IEEE International Symposium on Software Metrics*, pages 292–303, Nov 1999.

[6] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proc. of the 7th IEEE International Working Conference on Reverse Engineering*, pages 98–107, Nov 2000.

[5] Renamed code clones are the ones including different user-defined names in variable name or method name.

[6] Levenshtein distance is a method for comparing string S_a and S_b based on the number of insertion, deletions, and substitutions required to transform S_a to S_b.

[7] M. Balint, T. Girba, and R. Marinescu. How developers copy. In *Proc. of the 14th IEEE International Conference on Program Conprehension*, pages 56–68, Jun 2006.

[8] I. Baxter, A. Yahin, L. Moura, M. Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. International Conference on Software Maintenance 98*, pages 368–377, Mar 1998.

[9] M. Bruntink, A. V. Deursen, T. Tourẃe, and R. V. Engelen. An evaluation of clone detection techniques for identifying crosscutting concerns. In *Proc. of the 20th IEEE International Conference on Software Maintenance*, pages 200–209, Sep 2004.

[10] 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*, pages 36–43, Oct 2002.

[11] Canna. http://canna.sourceforge.jp/.

[12] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proc. of the 29th International Conference on Software Engineering*, pages 158–167, May 2007.

[13] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proc. of the International Conference on Software Maintenance 99*, pages 109–118, Aug 1999.

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

[15] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Aries: Refactoring support environment based on code clone analysis. In *Proc. of 8th IASTED International Conference on Software Engineering and Applications*, pages 222–229, Nov 2004.

[16] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Method and implementation for investigating code clones in a software system. *Information and Software Technology*, 49(9-10):985–998, Sep 2007.

[17] IEEE. *Standard for Software Maintenance.* IEEE Standard 1219, 1998.

[18] ISO/IEC. *Software Engineering - Software Maintenance.* ISO/IEC 14764, 1999.

[19] S. Jarzabek and L. Shubiao. Eliminating redundancies with a "composition with adaptation" metaprogramming technique. In *Proc. of ESEC-FSE'03 European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 237–246, Sep 2003.

[20] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, Jul 2002.

[21] C. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful. In *Proc. of the 13th Working Conference on Reverse Engineering - Volumn 00*, 2006.

[22] 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*, pages 83–92, Aug 2004.

[23] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 187–196, Sep 2005.

[24] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proc. of the 8th International Symposium on Static Analysis*, pages 40–56, Jul 2001.

[25] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transaction on Software Engineering*, 32(3):176–192, Mar 2006.

[26] Z. A. Mann. Three public enemies: Cut, copy, and paste. *IEEE Computer*, 39(7):31–35, Jul 2006.

[27] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proc. of the International Conference on Software Maintenance 96*, pages 244–253, Nov 1996.

[28] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Proc. of the 8th IEEE International Software Metrics Symposium*, pages 87–94, Jun 2002.

[29] S. Pfleeger. *Software Engineering: Theory and Practice.* Prentice Hall, 1998.

[30] M. Toomim, A. Begel, and S. Graham. Managing duplicated code with linked editing. In *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 173–180, Sep 2004.

[31] S. W. L. Yip and T. Lam. A software maintenance survey. In *Proc. of the 1st Asia-Pacific Software Engineering Conference*, pages 70–79, Dec 1994.