Enhancement of CRD-Based Clone Tracking

Yoshiki Higo, Keisuke Hotta, Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University, JAPAN
{higo,k-hotta,kusumoto}@ist.osaka-u.ac.jp

ABSTRACT

Many researchers have conducted a variety of research related to clone evolution. In order to grasp how clones have evolved, clones must be tracked. However, conventional clone tracking techniques are not feasible to track clones if they moved to another location in the source code. Consequently, in this research, we propose a new clone tracking technique. The proposed technique is an enhanced version of clone tracking with clone region descriptor (CRD) proposed by Duala-Ekoko and Robillard. The proposed technique can track clones even if they moved to another location. We have implemented a software tool based on the proposed technique, and applied it to two open source systems. In the experiment, we confirmed that the proposed technique could track 44 clone groups, which the conventional CRD tracking could not track. The accuracy of the tracking for those clones was 91%.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Design, Measurement

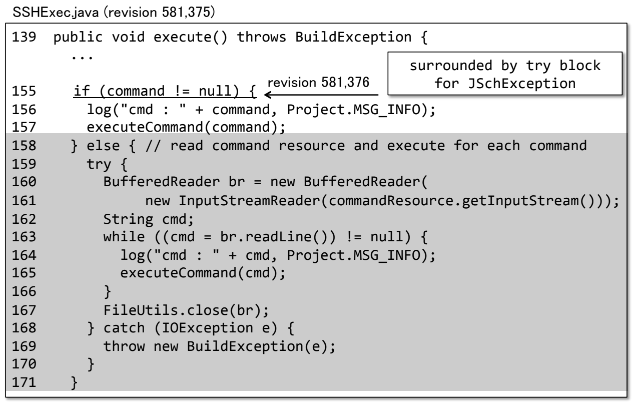
Keywords

Code clone, clone evolution, tracking clones

1. INTRODUCTION

Code cloning is a common practice in software development [13]. There are various reasons why clones occur in source code, for example, rapid implementation of required functionality, reuse of reliable existing code, and using existing code as templates for new functionality.

At the same time, there are many previous works saying that the presence of clones has negative impacts on software development [1, 11]. For example, if we modify a code fragment that has clones, we need to check whether each of the clones has to be modified simultaneously in the same way or not [5]. If clones to be modified are not modified, unintended inconsistencies occur in source code, and they may result in failures [8, 17].


SSHExec.java (revision 581,375)
revision 581,376
surrounded by try block for JSchException

```
public void execute() throws BuildException {
  ...
  if (command != null) {
    log("cmd : " + command, Project.MSG_INFO);
    executeCommand(command);
  } else { // read command resource and execute for each command
    try {
      BufferedReader br = new BufferedReader(
          new InputStreamReader(commandResource.getInputStream()));
      String cmd;
      while ((cmd = br.readLine()) != null) {
        log("cmd : " + cmd, Project.MSG_INFO);
        executeCommand(cmd);
      }
      FileUtils.close(br);
    } catch (IOException e) {
      throw new BuildException(e);
    }
  }
```

Figure 1: Actual modification that conventional techniques cannot track clones

By managing clones appropriately, which means maximizing positive aspects of cloning and minimizing its negative aspects, we can develop software systems efficiently [19]. In order to conduct and continue appropriate clone management, we need to have complete knowledge of clones. In the past, several research efforts have investigated occurrences and evolution of clones [12, 14, 16, 19].

However, conventional techniques are not feasible to track clones in the following cases even if they are still duplicated to one other.

- They are moved to other locations in the source code.
- They are changed consistently in a large way.

Figure 1 shows an actual clone that Duala-Ekoko et al.'s technique cannot track. The hatched part is the clone. In revision 581,376, a try-block for catching JSchException was added outside of the clone. In their technique, blocks are tracked based on the nesting structure. Hence, their technique cannot regard that the else-block after the modification corresponds to the else-block inside the new try-block. As a result, the clone is considered to have disappeared.

There are different kinds of approaches for tracking clones. For example, tracking clones by monitoring clipboard activity is a reasonable choice to manage clones [18]. Monitoring clipboard allows us to know when and where code was copied and pasted. However, we have to use a monitoring tool during software development and maintenance for tracking code clones.

Tracking clones is a fundamental technique in the field of software evolution, and it is used in a variety of research related to software evolution. Consequently, accurate and scalable clone tracking techniques are essential for such research. In this paper, we propose a new technique for clone tracking.

In this paper, we evaluate the accuracy of clone tracking of the proposed technique, and we show its application result for revealing why clones disappear during software evolution.

The major contributions of this paper are as follows:

- We have proposed a new technique for clone tracking. The proposed technique can track clones even if they are moved to another location in the source code.
- We have developed a clone tracking system based on the proposed technique. The system includes hash-based incremental clone detection function for realizing rapid clone detection. Currently, the system can handle Java source code and SVN repositories; however, it is not difficult to extend it to handle other programming languages and other repositories.
- We have compared the accuracy of tracking clones between the proposed technique and a conventional technique, which was proposed by Duala-Ekoko and Robillard [4]. We checked manually 44 clones that only the proposed technique could track, and we found that 40 out of them were correct tracking. We also investigated 61 clones that the proposed technique could not track, and we found that 56 out of them had actually disappeared.
- We have applied the proposed technique to OSS for revealing why clones disappear during software evolution. This is the first study focusing on reasons for clone disappearance.

The remainder of this paper is organized as follows: Section 2 describes related research; Section 3 explains the proposed technique, and Section 4 introduces a software tool that we have implemented; in Section 5, we evaluate the accuracy of clone tracking with the proposed technique; Section 6 shows an experimental result that the proposed technique was applied to open source systems for revealing why clones disappear; Section 7 discusses threats to validity of the experiment; Section 8 concludes this paper.

2. RELATED WORK

There are many studies that have investigated how clones evolve [2, 3, 10, 14, 15]. They empirically reveal phenomena and characteristics related to clone evolution. Their reports focus on investigating how clones have evolved, not on tracking clones themselves. The tracking techniques that they used were not feasible in the case where the code is moved. That means, in their investigations, tracking clones might not have been so accurate. If the proposed technique had been used in their investigations, the results might have changed.

Currently, there are several techniques for tracking clones. They can be classified into three categories. Techniques in the first category detect clones from every revision, then they link clones between every pair of consecutive revisions [3, 6, 14]. Those techniques use various heuristics for linking clones. If the amount of modification is not small, linking clones does not work well.

Techniques in the second category detect clones from the initial revision, then track the clones using change histories stored in software repositories [2, 15, 20]. However, those techniques do not consider clones that appear during the target period, so that they are not tracked.

Techniques in the third category are hybrid approaches between the first category and the second one [4]. Those techniques seek to merge approaches so as to complement the weak points of techniques in each category.

```
<CRD> ::= <file> <class> <CM> [<method>]
<method> ::= <signature> <block>*
<block> ::= <btype> <anchor> <CM>
<btype> ::= 'for' | 'while' | 'do' | 'if' | 'else' | 'switch' |
            'try' | 'catch' | 'finally' | 'synchronized'
```

(a) Definition

```
public class DeleteManager {
  ...
    public void delete (int n) {
      ...
        for(int i = n ; i < delete.size() ; i++ ) {
          ...
            if (delete.get(i) instanceof ElementNode) {
                // some code
            }                                              A
        ...
```

(b) Source Code

```
packagename.DeleteManager.java,DeleteManager,5
delete(int),5
for,delete.size(),4
if,delete.get(i) instanceof ElementNode,2
```

(c) Example

Figure 2: Clone Region Descriptor

However, even those techniques are not feasible to track clones in the case that clones are moved to other locations in the source code. Moreover, they take much time to track clones because they analyze all the files to detect clones for every of the target revisions.

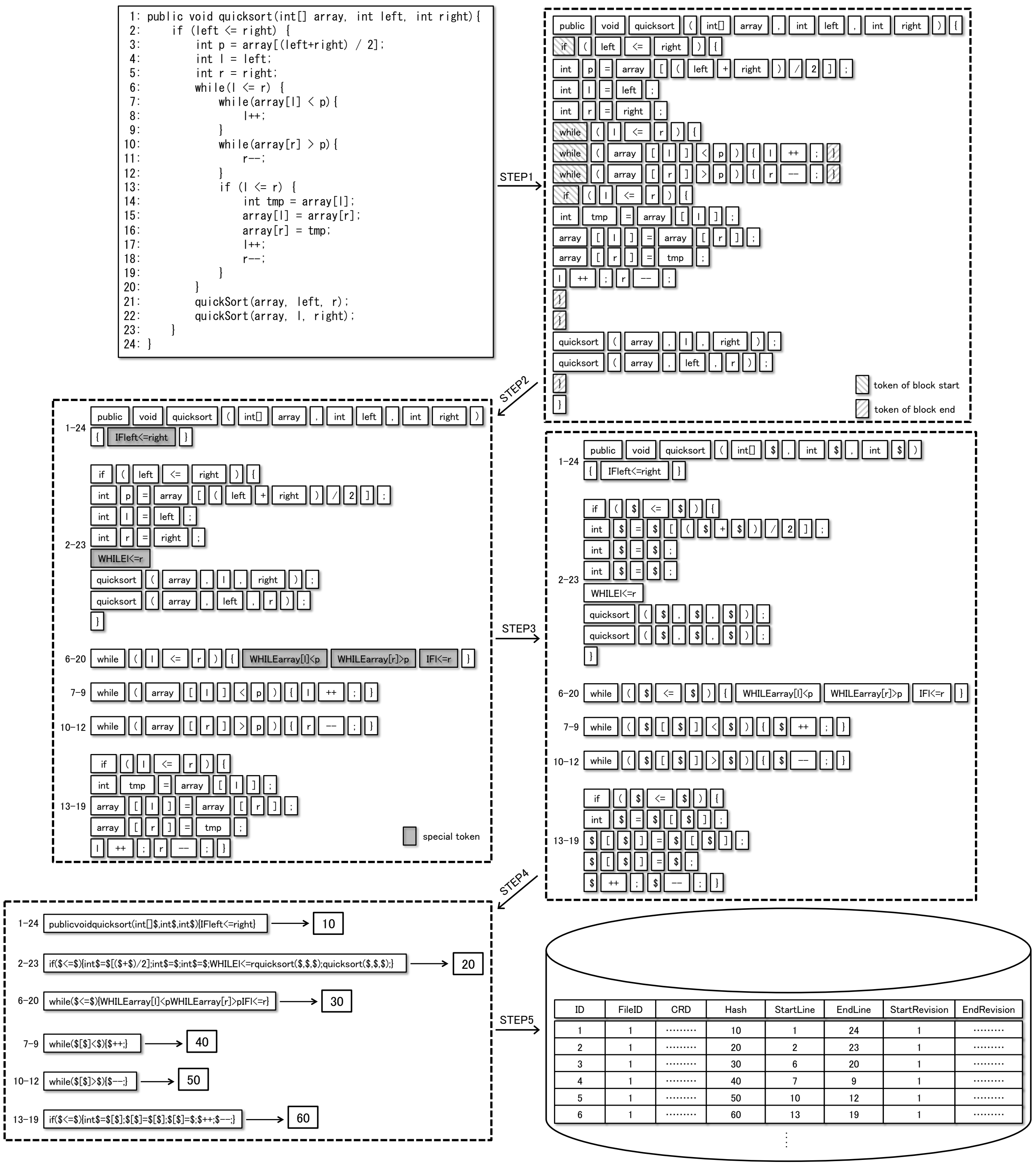
In this research, we would like to track clones through several thousand or more revisions, so that rapidity of tracking is important. Consequently, we develop a new clone tracking technique, which can complete tracking from several thousand of revisions within a couple of hours. The proposed technique uses CRDs (Clone Region Descriptor) which were proposed by Duala-Ekoko and Robillard [4]. In the proposed technique, we measure the similarity of CRD among revisions whereas they used exact matches of CRD. In the proposed technique, even if CRDs of two consecutive revisions are not exactly the same, they are linked if they are similar to one another. Linking clones with looser conditions allows them to be tracked even if they are moved to another location in source code.

3. TRACKING CLONES

Herein, we propose a new technique for tracking clones. The primary requirements for tracking clones are *accuracy* and *speed*.

- In order to achieve high speed tracking, we use an incremental hash-based clone detection. In an incremental detection, only the updated files are analyzed in every revision [7]. Higo et al. reported that the average number of updated files was only 3.53 in their experiment [9]. Hence, incremental detection is much more rapid than non-incremental one.
- In order to achieve high accuracy tracking, clones are tracked based on the similarity of their CRDs. The original CRD-based tracking proposed by Duala-Ekoko and Robillard links clones between a pair of consecutive revisions if and only if their CRDs are exactly the same. However, the condition is too strict to track code clones if they are moved to another location in source code. Consequently, we allow clones to be tracked even if their CRDs are different to a certain extent.

Our technique consists of two procedures, **hash generation** and **clone linking**, each of which is described briefly as follows.



```
 1: public void quicksort(int[] array, int left, int right){
 2:     if (left <= right) {
 3:         int p = array[(left+right) / 2];
 4:         int l = left;
 5:         int r = right;
 6:         while(l <= r) {
 7:             while(array[l] < p){
 8:                 l++;
 9:             }
10:             while(array[r] > p){
11:                 r--;
12:             }
13:             if (l <= r) {
14:                 int tmp = array[l];
15:                 array[l] = array[r];
16:                 array[r] = tmp;
17:                 l++;
18:                 r--;
19:             }
20:         }
21:         quickSort(array, left, r);
22:         quickSort(array, l, right);
23:     }
24: }
```

```
1-24   publicvoidquicksort(int[]$,int$,int$){IFleft<=right}  → 10
2-23   if($<=$){int$=$[($+$)/2];int$=$;int$=$;WHILEl<=rquicksort($,$,$);quicksort($,$,$);}  → 20
6-20   while($<=$){WHILEarray[l]<pWHILEarray[r]>pIFl<=r}  → 30
7-9    while($[$]<$){$++;}  → 40
10-12  while($[$]>$){$--;}  → 50
13-19  if($<=$){int$=$[$];$[$]=$[$];$[$]=$;$++;$--;}  → 60
```

ID	FileID	CRD	Hash	StartLine	EndLine	StartRevision	EndRevision
1	1	………	10	1	24	1	………
2	1	………	20	2	23	1	………
3	1	………	30	6	20	1	………
4	1	………	40	7	9	1	………
5	1	………	50	10	12	1	………
6	1	………	60	13	19	1	………

⋮

Figure 3: Intuitive example showing how hash values are measured from source code

HASH GENERATION A hash value is calculated from every block in source files in every revision. If two or more blocks in a revision have the same hash value, they are regarded as clones. All the hash values are stored into a database with their locational information such as file path, start line, end line, and revision. Also, a CRD [4] is measured for every block, and it is also stored into the database.

CLONE LINKING Cloned blocks in every revision are linked to blocks in the next revision based on the similarity of their CRDs. Then, their hash values are checked: if two blocks having the same hash value (h_r) in revision r have also the same hash value (h_{r+1}) in revision $r+1$, they are regarded as keeping a clone relationship during r and $r+1$ even if h_{r+1} is different from h_r.

In the remainder of this section, firstly we introduce CRDs in Subsection 3.1. Secondly, we describe procedure "*hash generation*" in Subsection 3.2. Finally, procedure "*clone linking*" is described in Subsection 3.3.

3.1 Clone Region Descriptor

A **Clone Region Descriptor** (**CRD**) is an abstract description of location of a clone region in a software system. A CRD is specified independently of the line number of the text in the source files. Figure 2 shows the definition and an example of a CRD[1]. Figure 2(a) shows the definition of CRD in the extended Backus-Naur form. Figure 2(c) shows an example of CRD, which represents block "**A**" in Figure 2(b). In the technique proposed by Duala-Ekoko and Robillard [4], if a block in revision r has exactly the same CRD as a block in revision $r+1$, they are regarded as the same block. On the other hand, we used the similarities of CRDs for clone tracking.

3.2 Hash Generation

The input of this procedure is a repository for a target system. The output is hash values of all the structural blocks in all the source files of all the target revisions. There are five steps in this procedure. Figure 3 shows how source files are handled at each step.

STEP1 (Syntactic Analysis) Performing syntactic analysis for input source files. Every source file is transformed into a sequence of tokens. At the same time, the start token and the end token of every block are identified. Only lexical analysis alone is not sufficient for identifying start and end tokens.

STEP2 (Splitting) Every subsequence corresponding to a block is extracted from the sequences of the source files. A special token is put into every of the extracted positions. A special token includes the following information:

- the type of the block (e.g., *if*, *while*, *for*),
- a conditional predicate, if the block is conditional one,
- its parameters, if the block is a method or a constructor.

STEP3 (Normalization) By using type information extracted in STEP1, variable names and literals are replaced with special tokens. Invoked method names and type names are not normalized.

STEP4 (Building block texts) A character sequence is built from every token sequence. A hash value is calculated from every one of the character sequences.

STEP5 (Persisting hash values) The hash values of all the blocks are stored into a database. At the same time, the file path, start line, end line, revision number, and the CRD of every block are stored.

The critical point for the procedure "*hash generation*" is STEP2, which is a set of operations in which every one of the sub-blocks in a block is extracted, then a small marker is put into every of the extracted positions. Those operations allow us to identify blocks including duplication as clones even if their sub-blocks are not duplicated. If a block is totally duplicated to another block, their hash values and all the hash values of their sub-blocks become the same.

3.3 Clone Linking

The input for this procedure is the output of the procedure "*hash generation*", namely hash values of all the blocks in all the target revisions. In this procedure, blocks in every revision are linked to blocks in the next revision.

Hash values of blocks are used for identifying clone relationships in every revision. Blocks having the same value are regarded as clones. In the remainder of this paper, we call a group of blocks having the same value an **Equivalent Block Group** (in short, **EBG**).

[1]These figures shows the same example used in the literature [4].

In this procedure, The proposed technique identifies corresponding blocks between revisions r and $r+1$ based on the similarities of their CRDs. If block b_r in revision r and block b_{r+1} in revision $r+1$ satisfy all the conditions, b_{r+1} is regarded as the corresponding block of block b_r.

CONDITION1 The type of b_{r+1} is the same as the one of b_r.

CONDITION2 If b_r and b_{r+1} are conditional blocks, their conditions are the same except variable names, method names and literals in them.

CONDITION3 If b_r and b_{r+1} are methods or constructors, their names are the same or their parameters are the same.

CONDITION4 In revision $r+1$, the CRD of b_{r+1} has the highest similarity with one of b_r.

CONDITION5 In revision r, the CRD of b_r has the highest similarity with that for b_{r+1}.

The CRD similarity between blocks b_r and b_{r+1} are measured by Levenshtein Distance (in short, LD). If the two blocks have the identical CRDs, the LD between them becomes 0, which is the minimum value. If the two blocks have completely different CRDs, the LD takes on its maximum value. If we consider the similarity of two CRDs, we can track clones even if they were moved.

Figure 4 shows examples of a revision history and the result of clone linking based on it. In Figure 4(a), the following modifications are performed.

- Method $b2$ in file *B.java* was changed ($r_1 \to r_2$)
- Method $a1$ in file *A.java* was changed ($r_2 \to r_3$)
- Method $b1$ in *B.java* was changed ($r_2 \to r_3$)
- Method $b2$ in *B.java* was moved to file *C.java* ($r_2 \to r_3$)
- Method $c1$ in *C.java* was deleted ($r_2 \to r_3$)
- Method $b1$ in *B.java* was changed ($r_3 \to r_4$)

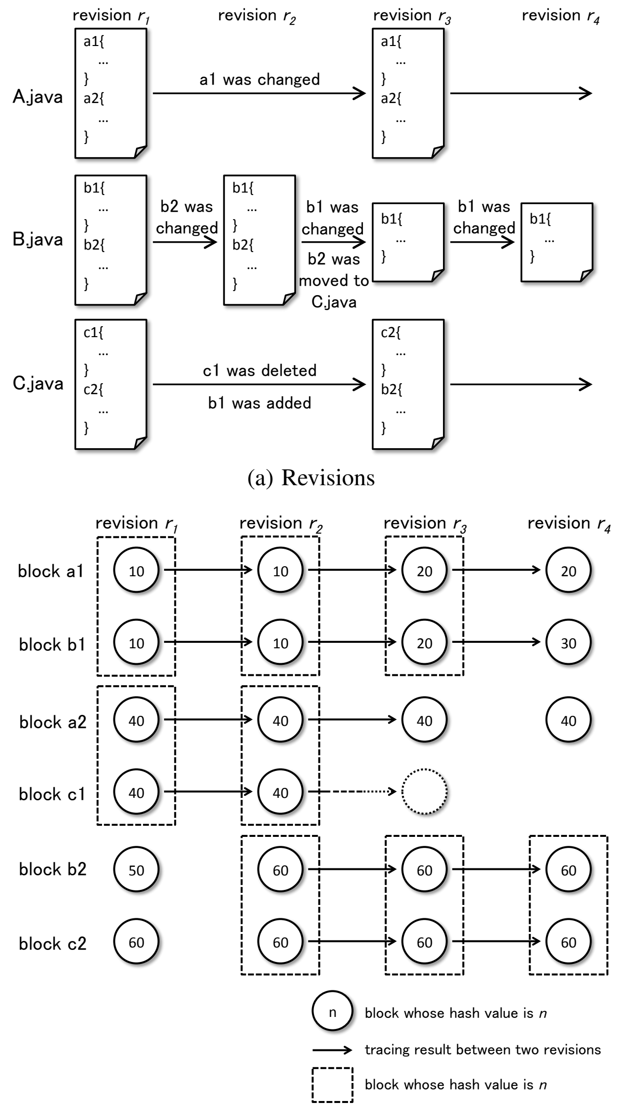
In Figure 4(b), each kind of objects has the following meanings.

- Every circle is a block. Its number is the hash value.
- Every arrow means a link of two blocks between consecutive two revisions.
- Every rectangle means an EBG. All the blocks in a rectangle have the same hash value.

Blocks a_1 and b_1 are modified between revisions r_2 and r_3, so that each of them has different hash values between the revisions. Even after the modification, they have the same hash value, which means that they continue to be duplicated in revision r_3. However, between revisions r_3 and r_4, only block b_1 is modified, and the two blocks have different hash values in revision r_4. Hence, the EBG consisting of a_1 and b_1 is regarded as disappeared in revision r_4.

Blocks a_2 and c_1 have the same hash value in revisions r_1 and r_2, which means they become clones in the revisions. However, in revision r_3, block c_1 itself disappears. In this case, the EBG consisting of a_2 and c_1 is regarded as disappeared in revision r_3.

Blocks b_2 and c_2 have the different hash values in revision r_1. Block b_2 is modified between revisions r_1 and r_2, so that they have the same hash value in revision r_2. Even block b_2 was moved to another file in revision r_3, it can be tracked. This is because the proposed technique tracks cloned blocks based on similarities of their CRDs.



(a) Revisions

(b) Tracking result

Figure 4: An example of clone tracking

After analyzing revision r_1

ID	FileID	CRD	Hash	StartLine	EndLine	StartRevisio	EndRevision
0	A.java	A.a1	10	1	10	r_1	r_4
1	A.java	A.a2	40	11	20	r_1	r_4
2	B.java	B.b1	10	1	10	r_1	r_4
3	B.java	B.b2	50	11	20	r_1	r_4
4	C.java	C.c1	40	1	10	r_1	r_4
5	C.java	C.c2	60	11	20	r_1	r_4

After analyzing revision r_2

ID	FileID	CRD	Hash	StartLine	EndLine	StartRevisio	EndRevision
0	A.java	A.a1	10	1	10	r_1	r_4
1	A.java	A.a2	40	11	20	r_1	r_4
2	B.java	B.b1	10	1	10	r_1	r_1
3	B.java	B.b2	50	11	20	r_1	r_1
4	C.java	C.c1	40	1	10	r_1	r_4
5	C.java	C.c2	60	11	20	r_1	r_4
6	B.java	B.b1	10	1	10	r_2	r_4
7	B.java	B.b2	60	11	20	r_2	r_4

After analyzing revision r_3

ID	FileID	CRD	Hash	StartLine	EndLine	StartRevisio	EndRevision
0	A.java	A.a1	10	1	10	r_1	r_2
1	A.java	A.a2	40	11	20	r_1	r_2
2	B.java	B.b1	10	1	10	r_1	r_1
3	B.java	B.b2	50	11	20	r_1	r_1
4	C.java	C.c1	40	1	10	r_1	r_2
5	C.java	C.c2	60	11	20	r_1	r_2
6	B.java	B.b1	10	1	10	r_2	r_2
7	B.java	B.b2	60	11	20	r_2	r_2
8	A.java	A.a1	20	1	10	r_3	r_4
9	A.java	A.a2	40	11	20	r_3	r_4
10	B.java	B.b1	20	1	10	r_3	r_4
11	C.java	C.c2	60	1	10	r_3	r_4
12	C.java	C.b2	60	11	20	r_3	r_4

After analyzing revision r_4

ID	FileID	CRD	Hash	StartLine	EndLine	StartRevisio	EndRevision
0	A.java	A.a1	10	1	10	r_1	r_2
1	A.java	A.a2	40	11	20	r_1	r_2
2	B.java	B.b1	10	1	10	r_1	r_1
3	B.java	B.b2	50	11	20	r_1	r_1
4	C.java	C.c1	40	1	10	r_1	r_2
5	C.java	C.c2	60	11	20	r_1	r_2
6	B.java	B.b1	10	1	10	r_2	r_2
7	B.java	B.b2	60	11	20	r_2	r_2
8	A.java	A.a1	20	1	10	r_3	r_4
9	A.java	A.a2	40	11	20	r_3	r_4
10	B.java	B.b1	20	1	10	r_3	r_3
11	C.java	C.c2	60	1	10	r_3	r_4
12	C.java	C.b2	60	11	20	r_3	r_4
13	B.java	B.b1	30	1	10	r_4	r_4

Figure 5: An example of Database Updating

4. IMPLEMENTATION

We developed a software tool, **CTEC**[2], based on the proposed technique. Currently, CTEC handles only Java language. However, it is not difficult to expand it to other programming languages because it performs only lexical and syntactic analyses as language dependent procedures. CTEC is written in Java and the language dependent procedure was implemented using Java Development Tools. Currently, CTEC uses method *java.lang.String.hashCode()* to generate hash values for blocks in the source code.

CTEC uses SQLite as its database module because of its ease of use. If we use another SQL database system such as PostgreSQL or the Oracle database, the speed of analysis would improve. However, we think that the speed with SQLite is sufficient.

In CTEC, for the two procedures "*hash generation*" and "*clone linking*" are implemented as independently. "*Hash generation*" registers hash values of blocks in the target revisions into an SQL-based database. "*Clone linking*" identifies a corresponding block in the next revision for every cloned block in every revision.

[2]The name came from an abbreviation of *CRD-based Tracker for Evolution of Clones*

Currently, we cannot find late propagation with CTEX because it tracks only duplicated blocks. If a block becomes unduplicated, it is removed from the set of tracking blocks. If we change CTEC to track not only duplicated blocks but also unduplicated blocks, we can find late propagation. However, tracking all the blocks will consume much more time.

The remainder of this section describes each of these two procedures, respectively.

4.1 Hash Generation

In order to achieve high scalability, we adopted an incremental hash generation. The following is an explanation of "*procedure for the 1st revision*" and "*procedure for the 2nd or later revisions*". In this explanation, we assume that the target revisions are $\{r_1, r_2, \cdots, r_n\}$.

In the procedure for the 1st revision, all the blocks in all the source files in r_1 are stored into an SQL database. Note that column "*EndRevision*" of all the blocks are set to r_n, which is the last revision of the target.

Table 1: Overview of Target Software

Software	Start revision (date)	End revision (date)	# of target revisions	LOC of start revision	LOC of end revision
ArgoUML	15,880 (2008-10-04)	19,794 (2011-11-17)	2,222	329,170	362,604
Ant	268,587 (2001-02-05)	904,537 (2010-01-30)	5,143	57,124	211,855

In the procedure for the 2nd or later revision, blocks in only source files modified, added, and deleted in r_k ($2 \leq k \leq n$) are stored into the database. If the database already includes blocks in the files, their columns "*EndRevision*" are updated to r_{k-1}. Note that if a file is modified, all the values for the blocks in the files are recalculated and stored again.

Herein, we explain how these procedures work with a simple example shown in Figure 5. This example shows how the database is updated for revisions shown in Figure 4. Figure 5 shows the database content after the procedure for every revision. Gray cells mean that they have just been inserted or updated. Note that, in this example, column "*FileID*" contains file names for ease to explain. However, in the actual implementation, column "*FileID*" includes IDs for source files. We have another database table for mapping file name and its ID.

In the procedure for revision r_1, which is the first revision of the target, all the source files are analyzed and their blocks are stored into the database. Note that column "*EndRevision*" of their blocks is r_4, which is the last revision of the target.

In the procedure for revision r_2, file *B.java* is reanalyzed, and all of its blocks are stored into the database. The database already has blocks in file *B.java*, and so their columns "*EndRevision*" are updated to r_1. This update operation is performed for only blocks whose "*EndRevision*" are r_4.

In revision r_3, files *A.java*, *B.java*, and *C.java* are modified. The procedure for those files in revision r_3 is performed as well as the procedure for file *B.java* in revision r_2.

4.2 Clone Linking

In order to link cloned blocks in every revision to corresponding blocks in the next revision (r_k and r_{k+1}), we need to obtain EBGs in revision r_k. Obtaining EBGs in revision r_k is performed by the following two steps.

STEP1 obtaining records (blocks) satisfying the following formula. This operation means obtaining all the blocks existing in r_k.

$$(StartRevision \leq r_k) \quad \bigwedge \quad (r_k \leq EndRevision) \qquad (1)$$

STEP2 classifying the blocks obtained in STEP1 based on their hash values. Two or more blocks having the same hash value form an EBG.

For each block in the EBGs identified in the STEP2, its corresponding block is found with the proposed technique described in Subsection 3.3.

5. EXPERIMENT

We conducted experiments on two well-known systems. The purpose of these experiments was to confirm that the proposed technique has a beneficial effect on clone tracking. In order to achieve the purpose, we have investigated tracking results by seeking to answer the following questions.

QUESTION1 Could the proposed technique track clones that the conventional technique could not track?

QUESTION2 Did clones that the proposed technique could not track really disappear?

Firstly, we describe the experimental setup, then we show the performance of CTEC. Lastly, we answer the questions.

Table 2: Timing information on experiment (with 8 threads)

Software	Hash generation (min.)	Clone linking total (min.)	max. (sec.)	min. (sec.)	ave. (sec.)
ArgoUML	132	43	46.0	0.043	21.3
Ant	100	50	31.2	2.8	17.1

5.1 Setup

In order to answer to the questions, we needed to track code clones with the proposed technique and a conventional technique. We used our tool, CTEC for tracking clones in both the ways. For tracking clones based on the original CRD-based way, CTEC was adjusted to track clones only if their CRDs are exactly the same. For tracking clones in the proposed way, CTEC was used as it was.

We selected Ant and ArgoUML as the targets of these experiments. Table 1 shows an overview of the target systems. They are managed by using SVN. The tracking targets are the source files under directories "*/ant/core/trunk/src/main*" and "*/trunk/src*", respectively. The source files under the directories comprise the "*trunk*", which is the main line of the development in SVN repositories. We also narrowed down to a subdirectory of "*trunk*" to exclude test files. The target period of the investigation was carefully chosen because we would like to investigate the development histories of multiple versions.

In this experiment, we specified 30 tokens as the threshold of minimum clone length. Thirty tokens is one of often-used thresholds in clone detection [11]. If a block includes at least 30 tokens, it can be output as a clone.

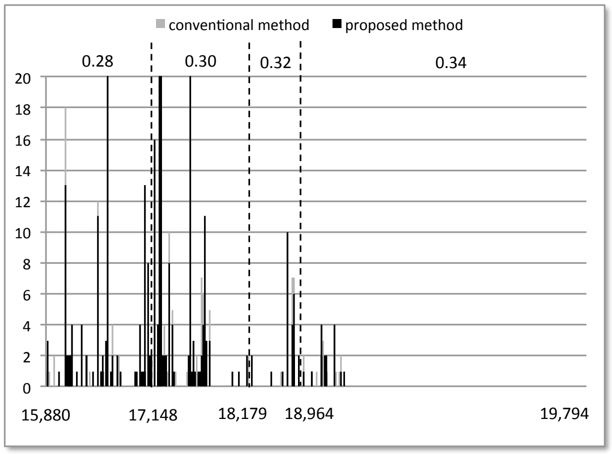
5.2 Performance

Table 2 shows the timing information of two procedures, "*hash generation*" and "*clone linking*". In this experiment, both of the procedures were performed with 8 threads and 4GBytes heap space[3]. CTEC took a couple of hours for "*hash generation*, which is a clone detection on all the target revisions. Besides, a "*clone linking*" task is performed on every pair of consecutive revisions. The table shows total, maximum, minimum, and average time for "*clone linking*" per pair. We can see that maximum time is 46 seconds, which means that we can perform a "*clone linking*" task interactively on demand from users. We can say that CTEC scales well enough for practical use.

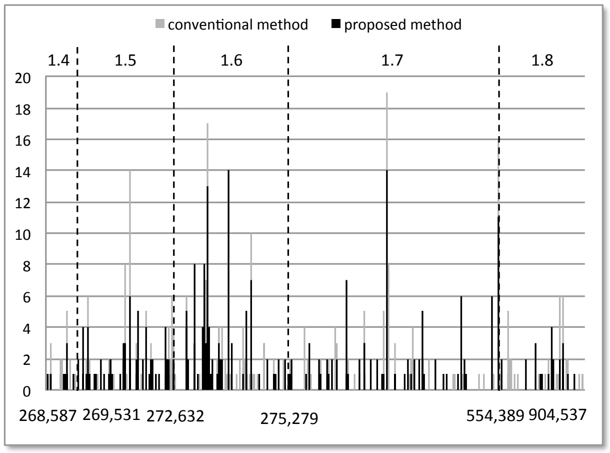
5.3 Answer to QUESTION1

We tracked clones by using the proposed technique and a conventional technique proposed by Duala-Ekoko and Robillard [4]. During the whole of the target period, the number of untrackable clones of the proposed technique and the conventional technique is 345 and 581 in Ant, and 537 and 739 in ArgoUML. There were not clones that the conventional technique tracked but the proposed

[3]The workstation used in the experiment has two octal-core CPUs. It is equipped with 128GBytes memory. The repositories and the databases lay on a SSD in the experiment.



(a) ArgoUML



(b) Ant

Figure 6: Number of blocks that were not tracked by either the proposed technique or the conventional one for every revision

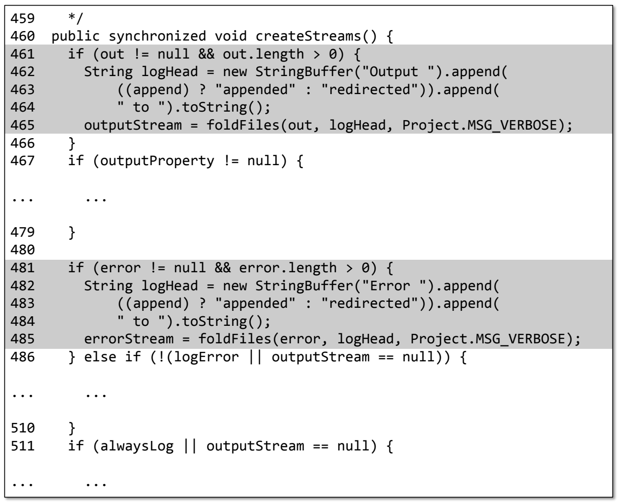
technique did not. That is, 236 and 202 clones were tracked only by the proposed technique, respectively.

Figure 6 shows the number of clones that were not tracked by the proposed or conventional techniques in every revision. Clones not tracked by the conventional technique are colored gray, and ones not tracked by the proposed technique are colored black. Black bars are drawn in front of gray bars. If a gray bar is taller than its corresponding black bar, there are clones that were tracked only by the proposed technique. The difference in length between the gray and black bars represents the number of such clones.

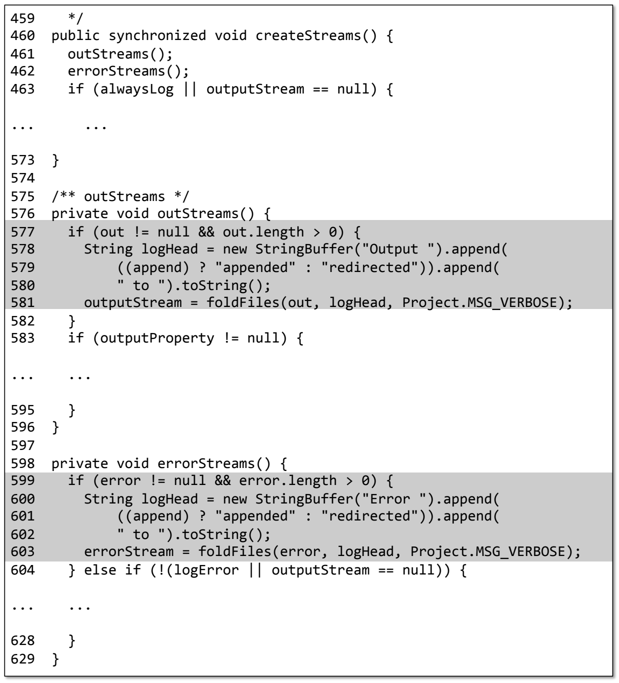
We investigated clones tracked only by the proposed technique to reveal whether tracking by the proposed technique had been correct or not. This was a manual investigation, so that we restricted the investigation to the period "1.8" in Ant and "0.32" in ArgoUML. In this investigation, we checked what kinds of modifications were performed in both the cases that tracking was correct and not correct. the following is a list of modifications. Prefix "**T**" means that tracking was correct even if the modifications were performed and "**F**" means that tracking was incorrect because of the modifications.

T1 Clones (and their surrounding code) were extracted as new methods.

T2 New blocks were added as surrounding code of clones such as null checking.


```
  */
public synchronized void createStreams() {
  if (out != null && out.length > 0) {
    String logHead = new StringBuffer("Output ").append(
        ((append) ? "appended" : "redirected")).append(
        " to ").toString();
    outputStream = foldFiles(out, logHead, Project.MSG_VERBOSE);
  }
  if (outputProperty != null) {

...   ...

  }

  if (error != null && error.length > 0) {
    String logHead = new StringBuffer("Error ").append(
        ((append) ? "appended" : "redirected")).append(
        " to ").toString();
    errorStream = foldFiles(error, logHead, Project.MSG_VERBOSE);
  } else if (!(logError || outputStream == null)) {

...   ...

  }
  if (alwaysLog || outputStream == null) {

...   ...
```

(a) Before modification (revision 567,592)


```
  */
public synchronized void createStreams() {
  outStreams();
  errorStreams();
  if (alwaysLog || outputStream == null) {

...   ...

}

/** outStreams */
private void outStreams() {
  if (out != null && out.length > 0) {
    String logHead = new StringBuffer("Output ").append(
        ((append) ? "appended" : "redirected")).append(
        " to ").toString();
    outputStream = foldFiles(out, logHead, Project.MSG_VERBOSE);
  }
  if (outputProperty != null) {

...   ...

  }
}

private void errorStreams() {
  if (error != null && error.length > 0) {
    String logHead = new StringBuffer("Error ").append(
        ((append) ? "appended" : "redirected")).append(
        " to ").toString();
    errorStream = foldFiles(error, logHead, Project.MSG_VERBOSE);
  } else if (!(logError || outputStream == null)) {

...   ...

  }
}
```

(b) After modification (revision 567,593)

Figure 7: An EBG that only the proposed technique tracked

T3 Conditional predicates were changed on conditional blocks including clones.

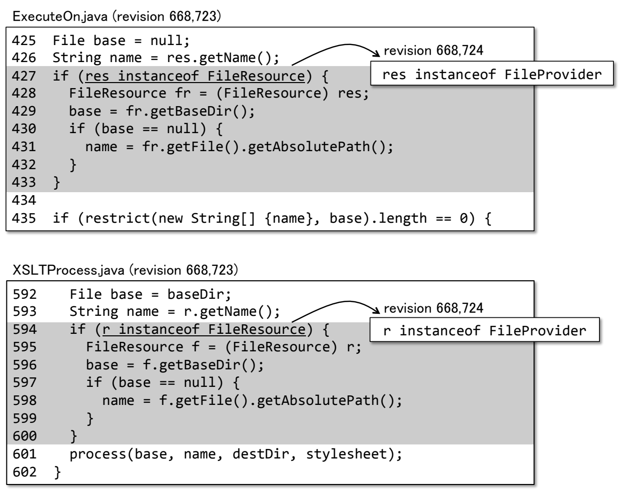
T4 Methods including clones were moved to other classes.

T5 New catch clauses were added on try blocks including clones. In the CRD definition of try block, it includes exception types of catch clauses attached to the try-block [4]. Consequently, if a new catch clause is added to a try block, the CRD of the try block changes.

T6 Methods were inlined as other methods.

F1 Cloned blocks were deleted. As a result, other blocks in the next revision were incorrectly linked to the deleted blocks.

F2 Cloned block became smaller than the threshold (30 tokens). Smaller blocks than the threshold were not registered to the


ExecuteOn.java (revision 668,723)
revision 668,724
res instanceof FileProvider
XSLTProcess.java (revision 668,723)
revision 668,724
r instanceof FileProvider

```
File base = null;
String name = res.getName();
if (res instanceof FileResource) {
  FileResource fr = (FileResource) res;
  base = fr.getBaseDir();
  if (base == null) {
    name = fr.getFile().getAbsolutePath();
  }
}

if (restrict(new String[] {name}, base).length == 0) {
```

```
  File base = baseDir;
  String name = r.getName();
  if (r instanceof FileResource) {
    FileResource f = (FileResource) r;
    base = f.getBaseDir();
    if (base == null) {
      name = f.getFile().getAbsolutePath();
    }
  }
  process(base, name, destDir, stylesheet);
}
```

Figure 8: Cloned blocks not tracked by the proposed technique because types in their conditions changed

database. They were treated as deleted blocks. As a result, incorrect linking occurred.

Figure 7 shows actual clones classified into **T1**. The cloned if-blocks and their subsequent code were extracted as new methods.

Table 3 shows the number of clones falling into each category. The number of correct tracking is 40, and the number of incorrect is only 4. That is, the accuracy of tracking for clones that were tracked by only the proposed technique is about 91%.

5.4 Answer to QUESTION2

We investigated whether clones not tracked by the proposed technique had really disappeared. We conducted a manual investigation on period "1.8" in Ant and "0.32" in ArgoUML as well as QUESTION1. As a result, we revealed the following modifications were factors in clones not being tracked by the proposed technique.

T1 Cloned blocks existed after modifications. However, their sizes became smaller than the threshold (30 tokens), so that they were considered to have disappeared by CTEC.

T2 Cloned blocks were deleted from the source code.

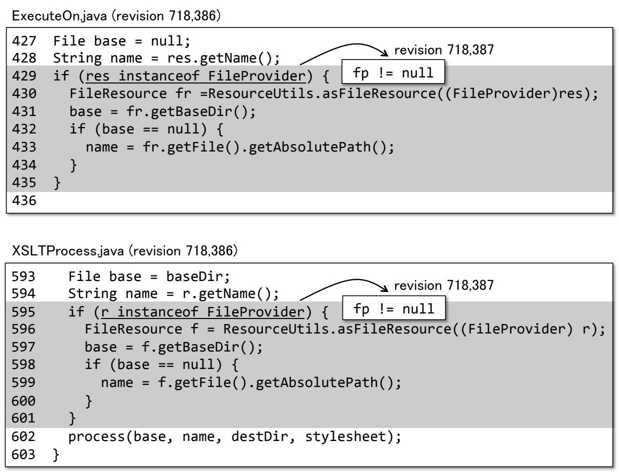
T3 Cloned blocks evolved to different code by large modifications.

F1 Types appearing in the conditions of cloned blocks were changed. In the proposed technique, variable names, method names, and literals are normalized but not types. Consequently, changes of types mean that CONDITION2 is unsatisfied.

F2 Conditions of cloned blocks were changed. This kind of change mean that CONDITION2 is unsatisfied.

Table 3: Clones tracked by only the proposed technique

	category		Ant	ArgoUML
T	tracking was appropriate		37	3
	T1	extracting as new methods	18	0
	T2	becoming deeper nested	9	0
	T3	changing block's conditions	5	2
	T4	moving methods	2	1
	T5	adding new catch clauses	2	0
	T6	in-lined to other methods	1	0
F	tracking was NOT appropriate		4	0
	F1	deleting blocks	3	0
	F2	shrinking blocks	1	0


ExecuteOn.java (revision 718,386)
revision 718,387
fp != null
XSLTProcess.java (revision 718,386)
revision 718,387
fp != null

```
File base = null;
String name = res.getName();
if (res instanceof FileProvider) {
  FileResource fr =ResourceUtils.asFileResource((FileProvider)res);
  base = fr.getBaseDir();
  if (base == null) {
    name = fr.getFile().getAbsolutePath();
  }
}

```

```
  File base = baseDir;
  String name = r.getName();
  if (r instanceof FileProvider) {
    FileResource f = ResourceUtils.asFileResource((FileProvider) r);
    base = f.getBaseDir();
    if (base == null) {
      name = f.getFile().getAbsolutePath();
    }
  }
  process(base, name, destDir, stylesheet);
}
```

Figure 9: Cloned blocks not tracked by the proposed technique because their conditions were changed

F3 New catch clauses were added to cloned try blocks. Such modifications mean that CONDITION2 is unsatisfied.

Figure 8 shows an actual instance of untracked clones because of a change in type in its condition (**F1**). In revision 688,724, FileResource was changed to FileProvider in the condition of the cloned block. If the proposed technique were to be designed to normalize types in the conditions of the clone blocks, this clones would tracked correctly. However, the more normalized conditions are, the more likely it is that blocks are probably tracked incorrectly.

Figure 9 shows another example of clones not being tracked by the proposed technique (**F2**). The change performed in this example is larger than the one in Figure 8. In order to track clones even if this kind of large modifications were performed on the conditions of the cloned block, CONDITION2 must become much weaker or even be removed. However, such changes on CONDITION2 will yield much more incorrect tracking. Consequently, tracking clones correctly even if their conditions are largely changed is not realistic for a CRD-based clone tracking approach.

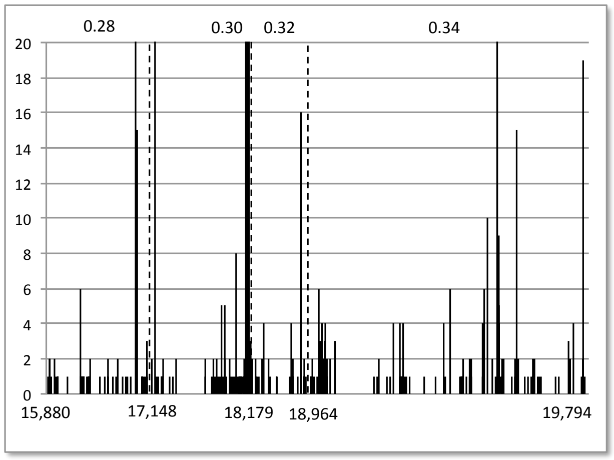
Table 4 shows the number of clones not tracked by the proposed technique because of such modifications. We manually investigated 61 untracked clones, and we found that 56 out of them had actually disappeared. That is, precision of our technique is about 92%.

6. INVESTIGATION ON WHY CLONES DISAPPEAR

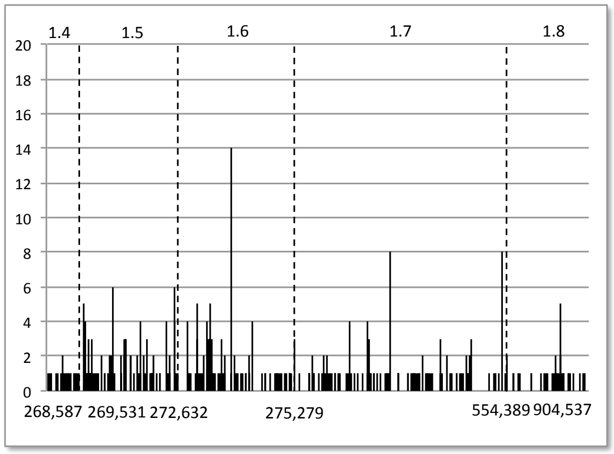
As an application of the proposed technique, we investigated why clones disappear during software evolution. In this application, we investigated why clone relationships among blocks had disappeared. On the other hand, the experiment described in Section 5 focuses on tracking each cloned block. In the past, several studies investigated occurrences and evolution of clones [12, 14,

Table 4: Clones not tracked by the proposed technique

	category		Ant	ArgoUML
T	not tracking was appropriate		23	33
	T1	shrinking blocks	7	9
	T2	deleting blocks	8	24
	T3	changing blocks	8	0
F	not tracking was NOT appropriate		5	0
	F1	changing types in conditions	2	0
	F2	changing conditions	2	0
	F3	adding catch clauses to try blocks	1	0



(a) ArgoUML



(b) Ant

Figure 10: Number of EBGs that disappeared or whose elements disappeared for every revision

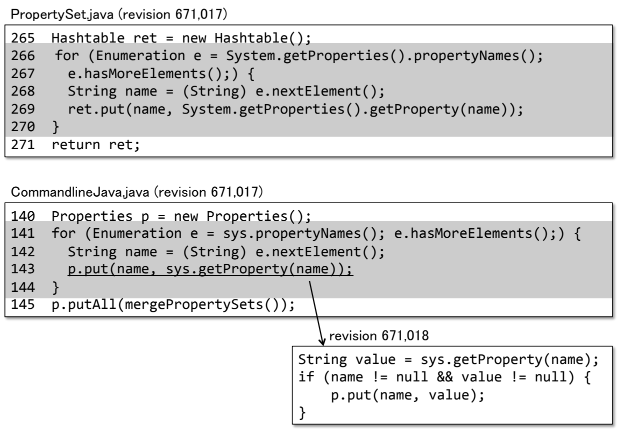
16, 19]: however, there is no research focusing on the investigation *why clones disappear*. Several empirical investigations on clones found that some of the detected clones disappeared during software evolution [14, 19]. However, in those investigations, clone removal is a by-product of clone evaluations. They did not investigate why clones had disappeared.

This investigation was also conducted on Ant and ArgoUML. Figure 10 shows the number of EBGs whose elements disappeared at each revision. As shown in this figure, clones disappear throughout the evolution of the software.

In order to reveal why clones disappear, we manually investigated why the EBGs disappeared. We investigated all the disappeared EBGs in periods "0.32" of ArgoUML (Figure 10(a)) and "1.8" of Ant (Figure 10(b)). The numbers of EBGs are 43 and 37,

Table 5: Reasons why clones disappeared

Reason	ArgoUML	Ant
Refactoring	10 (9)	7 (3)
Different evolution	6	11
Unintended inconsistency	15	10
Unneeded code deletion	8	5
Shrinking	4	0
CRD limitation	0	3
Total	43	36


PropertySet.java (revision 671,017)
CommandlineJava.java (revision 671,017)
revision 671,018

```
Hashtable ret = new Hashtable();
 for (Enumeration e = System.getProperties().propertyNames();
   e.hasMoreElements();) {
   String name = (String) e.nextElement();
   ret.put(name, System.getProperties().getProperty(name));
 }
return ret;
```

```
Properties p = new Properties();
for (Enumeration e = sys.propertyNames(); e.hasMoreElements();) {
  String name = (String) e.nextElement();
  p.put(name, sys.getProperty(name));
}
p.putAll(mergePropertySets());
```

```
String value = sys.getProperty(name);
if (name != null && value != null) {
    p.put(name, value);
}
```

Figure 11: Code where an unintended inconsistency occurred

respectively. Table 5 summarizes the results of the investigation.

The number of EBGs that disappeared due to *refactoring* are 10 and 7, respectively. However, some of those refactorings were not intended for removing duplicate code. We found that the other intentions were shortening long methods or simplifying complicated methods. Most of the *refactoring*s were of the *Extract Method* pattern. As a result of the refactorings, CONDITION2 became unsatisfied, so that EBGs could no longer be tracked.

Different evolution means that, different modifications (e.g., functionality enhancements or expansions) were applied to one or more blocks in an EBG, so that they evolved differently. We classified 6 and 11 EBGs into this category, respectively.

Unintended inconsistency means that clones disappeared unintentionally. For example, incomplete simultaneous modifications for a bug fix or for error checking were classified in this way. In this investigation, 15 and 10 EBGs were classified into this category. Figure 11 shows actual code of this category. This EBG consists of two blocks, which are in different source files. Only one of them was modified (*null* checking code was added) in revision 671,018. However, those two blocks are logically the same. The *null* checking should also be added to the other code.

Unneeded code deletion means EBGs were removed by deleting unneeded code. We investigated commit logs for deciding whether the commits were for deleting unneeded code or not. We classified 8 and 5 EBGs into this category.

Shrinking means the size of blocks in EBGs becomes smaller than the threshold of minimum clone size to be detected. All the blocks consisting of an EBG continue to be duplicated: however their size became smaller than the threshold as a result of consistent modifications. In consequence, they are no longer detected as clones after the modifications. In this investigation, 4 EBGs in ArgoUML were classified into this category.

CRD limitation means EBGs are judged to have disappeared because tracking was performed incorrectly. In this investigation, 3 EBGs in Ant were classified into this category.

7. THREATS TO VALIDITY

7.1 EBGs Categorization

In this experiment, we conducted manual investigations on open source systems. However, the investigation result may not be entirely correct because the authors are not developers of the target systems. In order to eliminate incorrectness as much as possible, two of the authors performed the investigation together. Totally, we spent approximately 10 hours for the categorizations.

7.2 Target Systems

In this experiment, we targeted only two system written in Java. Currently, it is difficult to generalize the investigation result because (1) only one programming language was investigated and (2) the number of investigated systems is only two. Furthermore, we selected ArgoUML and Ant as our targets because they are popular and successful systems. If we had selected other systems that are no more than moderately successful, the investigation results might have been different from this experiment.

8. CONCLUSION

This paper proposed a technique for tracking clones in software evolution. The proposed technique is an enhanced version of CRD-based clone tracking. The proposed technique includes incremental hash-based clone detection for rapid clone tracking. We conducted experiments on open source systems and confirmed the following.

- The proposed technique tracked many clones not tracked by a conventional technique due to Duala-Ekoko and Robillard. The accuracy percentage of tracking such clones was 91%.

- In the experiment, many clones were not tracked by even the proposed technique. However, most of such clones had actually become unduplicated by intended or unintended inconsistent modifications. The accuracy percentage of stopping tracking such clones of the proposed technique was 92%.

Moreover, we employed the proposed technique to investigate why clones disappear. We revealed that refactoring, different evolution, and unintended inconsistencies are major factors for clone disappearance. Interestingly, some of the refactorings were not intended for removing duplicate code but for shortening long methods or simplifying complicated methods.

In the future, we are going to replicate some empirical experiments that included clone tracking. Because the proposed technique can track clones more accurately than conventional techniques, we might obtain new findings.

9. ACKNOWLEDGEMENTS

This work was supported by MEXT/JSPS KAKENHI 25220003, 24650011, and 24680002.

10. REFERENCES

[1] G. Antoniol, U. Villano, E. Merlo, and M. Di Penta. Analyzing cloning evolution in the linux kernel. *Information and Software Technology*, 44:755–765, 2002.

[2] L. Aversano, L. Cerulo, and M. Di Penta. How clones are maintained: An empirical study. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 81–90, 2007.

[3] T. Bakota, R. Ferenc, and T. Gyimothy. Clone smells in software evolution. In *Proceedings of the 23rd International Conference on Software Maintenance*, pages 24–33, 2007.

[4] E. Duala-Ekoko and M. P. Robillard. Clone region descriptors: Representing and tracking duplication in source code. *ACM Transactions on Software Engineering and Methodology*, 20(1):3:1–3:31, 2010.

[5] R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger. Relation of code clones and change couplings. In *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering*, pages 411–425, 2006.

[6] N. Göde. Evolution of type-1 clones. In *Proceedings of the 9th International Working Conference on Source Code Analysis and Manipulation*, pages 77–86, 2009.

[7] N. Göde and R. Koschke. Studying clone evolution using incremental clone detection. *Journal of Software: Evolution and Process*, 25(2):165–192, 2013.

[8] Y. Higo and S. Kusumoto. How often do unintended inconsistencies happen? –deriving modification patterns and detecting overlooked code fragments–. In *Proceedings of the 28th International Conference on Software Maintenance*, pages 222–231, 2012.

[9] Y. Higo, U. Yasushi, M. Nishino, and S. Kusumoto. Incremental code clone detection: A PDG-based approach. In *Proceedings of the 18th Working Conference on Reverse Engineering*, pages 3–12, 2011.

[10] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto. Is duplicate code more frequently modified than non-duplicate code in software evolution?: an empirical study on open source software. In *Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, pages 73–82, 2010.

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

[12] C. Kapser and M. W. Godfrey. "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.

[13] M. Kim, L. Bergman, T. Lau, and D. Notkin. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering*, pages 83–92, 2004.

[14] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held jointly with the 13th International Symposium on Foundations of software engineering*, pages 187–196, 2005.

[15] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Proceedings of the 14th Working Conference on Reverse Engineering*, pages 170–178, 2007.

[16] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, pages 492–501, 2006.

[17] 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*, 32(3):176–192, 2006.

[18] M. de Wit, A. Zaidman, and A. van Deursen. Managing code clones using dynamic change tracking and resolution. In *Proceedings of the 25th International Conference on Software Maintenance*, pages 169–178, 2012.

[19] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Clone management for evolving software. *IEEE Transactions on Software Engineering*, 38(5), 2012.

[20] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15(1):1–34, 2010.