Scalable Detection of Semantic Clones Based on Two-Stage Clustering

Shunsuke Yoshioka, Norihiro Yoshida, Kyohei Fushida, Hajimu Iida
Graduate School of Information Science, Nara Institute of Science and Technology, Japan
Email: {shunsuke-y, yoshida, kyohei-f}@is.naist.jp, iida@itc.naist.jp

***Abstract*—We propose a scalable technique to detect semantic clones (i.e. semantically identical or similar code fragments) based on two-stage clustering, and show an example of detected clones.**

***Keywords*-Clone Detection, Clustering, Copy-and-paste**

I. Introduction

In practical software development, much duplicated code fragments are developed because of the ease of "copy-and-paste" of code portions. Modifications of these copied code fragments now makes these fragments "similar" and no longer identical. This can cause inconsistencies in modifications because it is difficult to find all similar fragments, especially in large scale source code.

There has been several studies that have been done on the automatic detection of semantically similar but not identical code fragments [1] [2]. However, these detection methods have proven to be time consuming.

In this paper, we propose a scalable technique that detects "semantic similar clone" (defined as follow: if pair of code fragments have similar control flow and have many overlapping statements, we regard them as "semantic similar clones"). We detected semantic simillar clones by using of the combination of coarse-grained and fine-grained clustering of feature vectors derived from a token type and syntax.

II. Algorithm Description

In this section, we illustrate our clone detection algorithm.

Firstly, code fragments are extracted from source code. Then, all of the extracted fragments are classified into clusters based on their characteristics. This step is divided into two stages. In the first stage, code fragments are coarsely classified in order to obtain good enough result in a short time. In the second stage, the results of the first stage are then finely classified to obtain more precise clusters. Finally, the resultant clusters are converted into a collection of clone sets (i.e., sets of semantically similar code fragments).

A. Extraction of Code Fragments

In this step, as many as possible code fragments are extracted for clustering. The size of a code fragment is fixed by a user-defined parameter w that defines the number of statements. Given that n is the number of statements in a method, then $n - w + 1$ code fragments are extracted from the method.

Code fragments cannot be properly extracted in a start/end of a block where '{' or '}' appears. This is because the 'block' encapsulates multiple statements and therefore is interpreted as a statement itself. For this reason, we redefine a statement as follows: '{' or '}' is treated as a statement itself, and the nature of the block which has an aspect of a statement is discarded. For example, from our definition, the following code fragment has four statements

```
if (len < 1024)
{
   write(msg[i].toString());
}
```

B. First Stage of Clustering

The purpose of the first stage is to obtain coarse-grained clusters within a short time-frame. Code fragments are represented as a feature vector to get the similality of the control flow. The components of the feature vector consist of: (1) whether code fragment contains `if, break, try, return` or some other types of token which relevant to control flow; and (2) an order of an appearance of '{' or '} in a code fragment.

C. Second Stage of Clustering

The purpose of the second stage is to obtain more precise clusters.

Code fragments detected in II-B are now reprensented as feature vectors by statement units (so that code fragments have w number of feature vectors) and reclassified according to the degree of overlapped feature vectors.

Formally let $\{C_i\}$ be a collection of clone set detected in II-B. Clone set C_i refers to the i-th element in the collection. Let $f_{i,j}$ be an element in clone set C_i. $f_{i,j}$ is a code fragment, thus referring to the j-th element in C_i. Let $s_{i,j,k}$ be a statement. $s_{i,j,k}$ is divided from code fragment $f_{i,j}$ with 1-statement increments, therefore the k-th split of $f_{i,j}$ (viz. $k \in \{1, 2, \dots, w\}$). $\{C_i\}$ are reclassified by a evaluation of similarity between code fragment $f_{i,x\in\{x|1,2,...\}}$ and $f_{i,y\in\{y|1,2,...\ and\ y\neq x\}}$. At first, statement $s_{i,x,1}$ is set to clustering query and is classified with statement set $\{s_{i,y,k}\}$. If more than one statement exist in the same cluster as statement $s_{i,x,1}$, then one of them is discarded from statement set $\{s_{i,y,k}\}$. Next, $s_{i,x,2}$, $s_{i,x,3}$,..., and $s_{i,x,w}$ is processed in a same way as $s_{i,x,1}$ have been done. Finally, if the number of remaining statements in $\{s_{i,y,k}\}$ is less

```
1  List paths = JarVerifier.convertCertsToChains(certChains);
2  boolean found = false;
3    for (Iterator t = paths.iterator() ; t.hasNext();)
4    {
5    X509Certificate [] path = (X509Certificate[])t.next();
6    X509Certificate cert = path[0] ;
7    if ( cert.equals(signer))
8    {
9      found = true;
10     break;
11   }
12 }
13 if (found == false)
14 {
15   throw new SecurityException("Jurisdiction policy files
          are " + "not signed by trusted signers!") ;
16 }
```

```
1  List signers = JarVerifier.getSignersOfJarEntry(
        jceCipherURL);
2  for (Iterator t = signers.iterator(); t.hasNext();)
3  {
4    X509Certificate [] chain = (X509Certificate[])t.next();
5    if ( chain[0].equals(jceCertificate))
6    {
7      signers = null;
8      break;
9    }
10 }
11 if (signers != null)
12 {
13   throw new SecurityException("Jurisdiction policy files
          are " + "not signed by trusted signers!");
14 }
15 CryptoPermissions defaultExport = new CryptoPermissions();
16 CryptoPermissions exemptExport = new CryptoPermissions();
```

Figure 1: An example of detected clones

Table I: Running time (sec) of Takana, DECKARD and Scorpio

	ant	jdtcore	swing	jdk1.4.2
Takana	3	46	30	42
DECKARD	-	-	-	142
Scorpio	34	2,283	142	-

than user defined threshold $\mathfrak{G}$, then the code fragment $f_{i,y}$ is evaluated as similar to $f_{i,x}$ and a clone of $f_{i,x}$.

III. Imprimentation and Evaluation

We implemented our proposed algorithm as a code clone detection tool called Takana. Using the $\mathfrak{G}$=1 setting, we evaluated Takana using the following projects: eclipse-ant, eclipse-jdtcore, j2sdk.1.4.0-swing, jdk (ver. 1.4.2) and jdk (ver.1.5.0). All projects consumed less that 2 GB of memory during the evaluation.

Figure 1 shows one of detected code clone in jdk 1.5.0. As seen in this figure, lines 1-16 on the left have a clone relation to lines 1-14 on the right.

Figure 2 shows the running time against the different sizes of w. The vertical axis shows the running time, while the horizontal axis represents w.

Figure 3 compares the running times against the size of the source code. The vertical axis represents the running time, and the horizontal being the size of the source code. Different colored points in Figure 3 represent the running time for each of the evaluated systems. Figure 3 indicates that running time has a constant increase.

Table I is comparison of a running time with other clone detection tools DECKARD [3] and Scorpio[4]. DECKARD is abstract syntax tree based tool and Scorpio is program dependence graphs based tool.

IV. Summary and Future work

We proposed a method to detect semantic clones based on two-stage clustering, and showed an example of detected clones. As future work, we are planning to perform a quantitative investigation of detected clones.

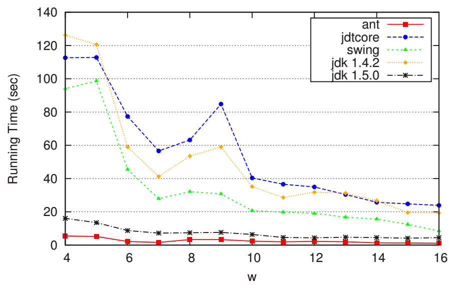

Figure 2: Running time under different size of w

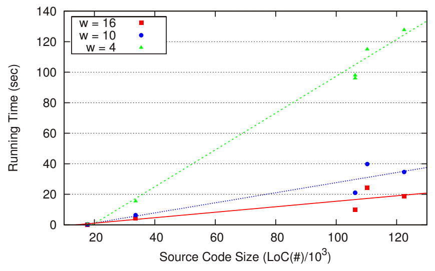

Figure 3: Running time under different size of source codes

Acknowledgment

We thank Mr. Raula Gaikovina Kula of Nara Institute of Science and Technology for proofreading this paper. This work is being conducted as a part of StagE Project, supported by MEXT, Japan. Also, this work was supported by KAKENHI (22500027, 22800040, 22800043).

References

[1] H. Kim, Y. Jung, S. Kim, and K. Yi, "MeCC: memory comparison-based clone detector," in *Proc. ICSE 2011*, 2011, pp. 301–310.

[2] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proc. ICSE 2008*, 2008, pp. 321–330.

[3] Y. Higo and S. Kusumoto, "Improvement and evaluation of code clone detection using program dependency graph," *IPSJ Journal*, vol. 51, no. 12, pp. 2149–2168, Dec 2010, in Japanese.

[4] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proc. ICSE 2007*, 2007, pp. 96–105.