Abstract—We propose a scalable technique to detect seman-
tic 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 make these fragments “similar” and no longer identical. This can cause inconsistencies in modifi-
cations 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 similar clones by using of 
the combination of coarse-grained and fine-grained cluster-
ing 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 clus-
ters 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 
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

```java
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 repre-
sented as a feature vector to get the similarity 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 represented 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 code 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. $k \in \{1, 2, \ldots, w\}$). $\{C_i\}$ are reclassified by a 
evaluation of similarity between code fragment $f_{i,x} \in \{x|1,2,\ldots\}$ 
and $f_{i,y} \in \{y|1,2,\ldots \}$. 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, $s_{i,x,2}, s_{i,x,3},\ldots$, 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
```java
List paths = JarVerifier.convertCertsToChains(certChains);
boolean found = false;
for (Iterator t = paths.iterator(); t.hasNext();)
{
    X509Certificate[] path = (X509Certificate[])t.next();
    X509Certificate cert = path[0];
    if (cert.equals(signer))
    {
        found = true;
        break;
    }
} if (found == false)
{
    throw new SecurityException("Jurisdiction policy files are not signed by trusted signers!");
}
```

```java
List signers = JarVerifier.getSignersOfJarEntry(jceCipherURL);
for (Iterator t = signers.iterator(); t.hasNext();)
{
    X509Certificate[] chain = (X509Certificate[])t.next();
    if (chain[0].equals(jceCertificate))
    {
        signers = null;
        break;
    }
} if (signers != null)
{
    throw new SecurityException("Jurisdiction policy files are not signed by trusted signers!");
}
```

```java
CryptoPermissions defaultExport = new CryptoPermissions();
CryptoPermissions exemptExport = new CryptoPermissions();
```

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

<table>
<thead>
<tr>
<th></th>
<th>ant</th>
<th>jdtcore</th>
<th>swing</th>
<th>jdk 1.4.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Takana</td>
<td>3</td>
<td>46</td>
<td>30</td>
<td>42</td>
</tr>
<tr>
<td>DECKARD</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>142</td>
</tr>
<tr>
<td>Scorpio</td>
<td>34</td>
<td>2.283</td>
<td>142</td>
<td>-</td>
</tr>
</tbody>
</table>

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

### III. IMPLEMENTATION AND EVALUATION

We implemented our proposed algorithm as a code clone detection tool called Takana. Using the $\delta=1$ 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 1:** An example 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**


