Code Clones

Spiros Mancoridis[1]
Modified by Na Meng

Overview

- Definition and categories
- Clone detection
- Clone removal refactoring

Code Clones

- Code clone is a code fragment in source files that is identical or similar to another
- Code clones are either within a program or across different programs
- Clone pair: two clones
- Clone class: a set of fragments which are clones to each other

Code Clone Categorization

- Type-1 clones
  - Identical code fragments but may have some variations in whitespace, layout, and comments
- Type-2 clones
  - Syntactically equivalent fragments with some variations in identifiers, literals, types, whitespace, layout and comments

Code Clone Categorization

- Type-3 clones
  - Syntactically similar code with inserted, deleted, or updated statements
- Type-4 clones
  - Semantically equivalent, but syntactically different code

Key Points of Code Clones

- Pros
  - Increase performance
    - Code inlining vs. function call
  - Increase program readability
- Cons
  - Increase maintenance cost
  - If one code fragment contains a bug and gets fixed, all its clone peers should be always fixed in similar ways.
  - Increase code size
Clone Detection Strategies
• Text matching
• Token sequence matching
• Graph matching

Text Matching
• Older, studied extensively
• Less complex, and most widely used
• No program structure is taken into consideration
• Type-1 clones & some Type-2 clones
• Two types of text matching
  – Exact string match
    • Diff (cvs, svn, git) is based on exact text matching
  – Ambiguous match

Ambiguous Match
• Longest Common Subsequence match
• N-grams match

Token Sequence Matching
• A little more complex, less widely used
• No program structure is taken into account, either
• Type-1 and Type-2 clones
• CCFinder[2]
• CP-Miner[3]

CCFinder
• Step 1: Convert a program with multiple files to a single long token sequence
• Step 2: Find longest common subsequence of tokens

Step 1: Tokenization

```cpp
int main() {
    int i = 0;
    static int j = 5;
    while (i < 20) { // [...]
        i = i + j;
    }
    std::cout << "Hello World" << std::endl;
    return 0;
}
```
Remove white spaces
Step 1: Tokenization

```java
int main(){
    int i = 0;
    while(i<40){
        int sum = 0;
        System.out.println("Hello World");
        int j = 10;
        int k = 20;
        int l = 30;
        int m = 40;
        sum += a+b+c;
        System.out.println("hello world");
    }
    return 0;
}
```

Step 1: Tokenization

```java
int main(){
    if (RE == 0)
        RE == 0;
    while (RE <= 20) {
        cout << "Hello World" < e << endl;
        return 0;
    }
}
```

Step 1: Tokenization

```java
int main(){
    RE = 0;
    while (RE <= 20) {
        cout << "Hello World" < e << endl;
        return 0;
    }
}
```

Step 2: Find Clones

Detected Clone Pair Example[2]

Limitations of CCFinder

- All files are converted into a long token sequence
  - When the program contains millions of lines of code, the tool cannot perform efficiently
- Do not take into account the natural boundary between functions and classes
**CP-Miner[3]**

- Cut the token sequences by considering basic blocks as cutting units
- Calculate a hashcode for each subsequence
- Compare hashcode sequences instead of the original token sequences

**Graph Matching**

- Newer, bleeding edge
- More complex
- Type-1, Type-2, and Type-3 clones
- Syntactic and semantic understanding
  - AST matching (ChangeDistiller)
  - CFG matching (JDiff[4])
  - PDG matching ([5])

**CFG-based Clone Detection[4]**

- A Differencing Algorithm for Object-Oriented Programs
  - Match declarations of classes, fields, and methods by name
  - Match content in methods by hammock graphs
    - A hammock is a single entry, single exit subgraph of a CFG

**Example: Enhanced CFG comparison for P and P’**

**Hammock Graph Creation**

<table>
<thead>
<tr>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> hammock node n, n’, look-ahead threshold LH</td>
</tr>
<tr>
<td><strong>Output:</strong> set of matched pairs N</td>
</tr>
<tr>
<td><strong>Algorithm</strong></td>
</tr>
<tr>
<td>1. expand n and n’ one level to graph G and G’</td>
</tr>
<tr>
<td>2. Push start node pair &lt;s, s’&gt; to stack ST</td>
</tr>
<tr>
<td>3. while ST is not empty</td>
</tr>
<tr>
<td>4. pop &lt;c, c’&gt; from ST</td>
</tr>
<tr>
<td>5. if c or c’ is already matched then</td>
</tr>
<tr>
<td>6. continue;</td>
</tr>
<tr>
<td>7. if &lt;c, c’&gt; does not match then</td>
</tr>
<tr>
<td>8. compare c with LH successors of c’ or compare c’ with LH successors of c until find match</td>
</tr>
<tr>
<td>9. if a match is found then</td>
</tr>
<tr>
<td>10. ( N = N \cup {c, c’, &quot;unchanged&quot;} )</td>
</tr>
<tr>
<td>11. else</td>
</tr>
<tr>
<td>12. ( N = N \cup {c, c’, &quot;modified&quot;} )</td>
</tr>
<tr>
<td>13. push the pair’s sink node pair on stack</td>
</tr>
</tbody>
</table>
Observations

- The look-ahead process is like bounded LCS algorithm
  - It can tolerate statement insertions at the same level
- The algorithm starts from the outmost Hammock, so it is similar to top-down tree-differencing algorithm
- When statements are inserted at the higher level, the algorithm does not work well
  - $<c, c', "modified">$

PDG-based Clone Detection [5]

- Using Slicing to Identify Duplication in Source Code
  - Step 1: Partition PDG nodes into equivalence classes based on the syntactic structure, such as while-loops
  - Step 2: For each pair of matching nodes $(r_1, r_2)$, find two isomorphic subgraphs containing $r_1$ and $r_2$

Algorithm to Find Isomorphic Subgraphs

1. Start from $r_1$ and $r_2$, use backward slicing in lock step to add predecessors iff predecessors also match
2. If two matching nodes are loops or if-statements, forward slicing is also used to find control dependence successors (statements contained in the structure)

Example

<table>
<thead>
<tr>
<th>Fragment 1</th>
<th>Fragment 2</th>
</tr>
</thead>
</table>
| ```
while (isalpha(c)) {
  c = 'i';
  if (p == token_buffer || p == token_buffer(p))
    p = grow_token_buffer(p);
  if (c == 'i') c = ' ';
  *p++ = c;
  c = getc(input);
}
``` | ```
while (isdigit(c)) {
  if (p == token_buffer || p == token_buffer(p))
    p = grow_token_buffer(p);
  x = x + x + x + x;
  c = getc(flip);  `}
|``` |

Observations

- Pros
  - Tolerate statement reordering and some program structure changes
- Cons
  - Expensive
    - Points-to analysis
  - Do not allow ambiguous match
Summary

• Clone detection flexibility
  – PDG > CFG|AST > Token > Text
• Cost
  – Text < Token < CFG|AST < PDG

Clone Removal Refactoring

• Extract method
  – Extract the common code from different methods and create a method for it
• Pull up method
  – Pull up the duplicated method to the super class, and declare a new super class if there is none

Reference

[4] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold, A Differencing Algorithm for Object-Oriented Programs, ASE’04