A Mutation / Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools

Chanchal K. Roy and James R. Cordy
School of Computing, Queen's University, Kingston, Ontario, Canada

Abstract

In this poster we present an automated method for empirically evaluating clone detection tools. Our method leverages mutation-based techniques to overcome existing limitations of tool evaluation studies by automatically synthesizing large numbers of known clones based on an editing theory of clone creation. Our framework is effective in measuring recall and precision of clone detection tools for various types of fine-grained clones in real systems without manual intervention.

1 Introduction

- Copying a code fragment and reusing it by pasting with or without minor modifications is a common practice in software development.
- Consequently, software systems often have duplicated code (between 7% and 23%) [2, 8].
- However, duplication is harmful for software maintenance and evaluation (e.g., bug propagation) [8].
- Thus, need to detect these "code clones".
- Fortunately, a great many tools have been proposed [7, 8].

Motivation:

- This huge number of tools calls for quantitative evaluations and there have been several.
- But there is a lack of a validated clone benchmark.
- And huge manual effort is required to hand check large numbers of candidate clones.
- Bellon et al. [3] is the most extensive one to-date, but only 2% of clones were oracled and many parameters may have influenced the results [2].
- Thus, we propose a controlled experiment, using the well known mutation analysis technique [1] from the testing community.

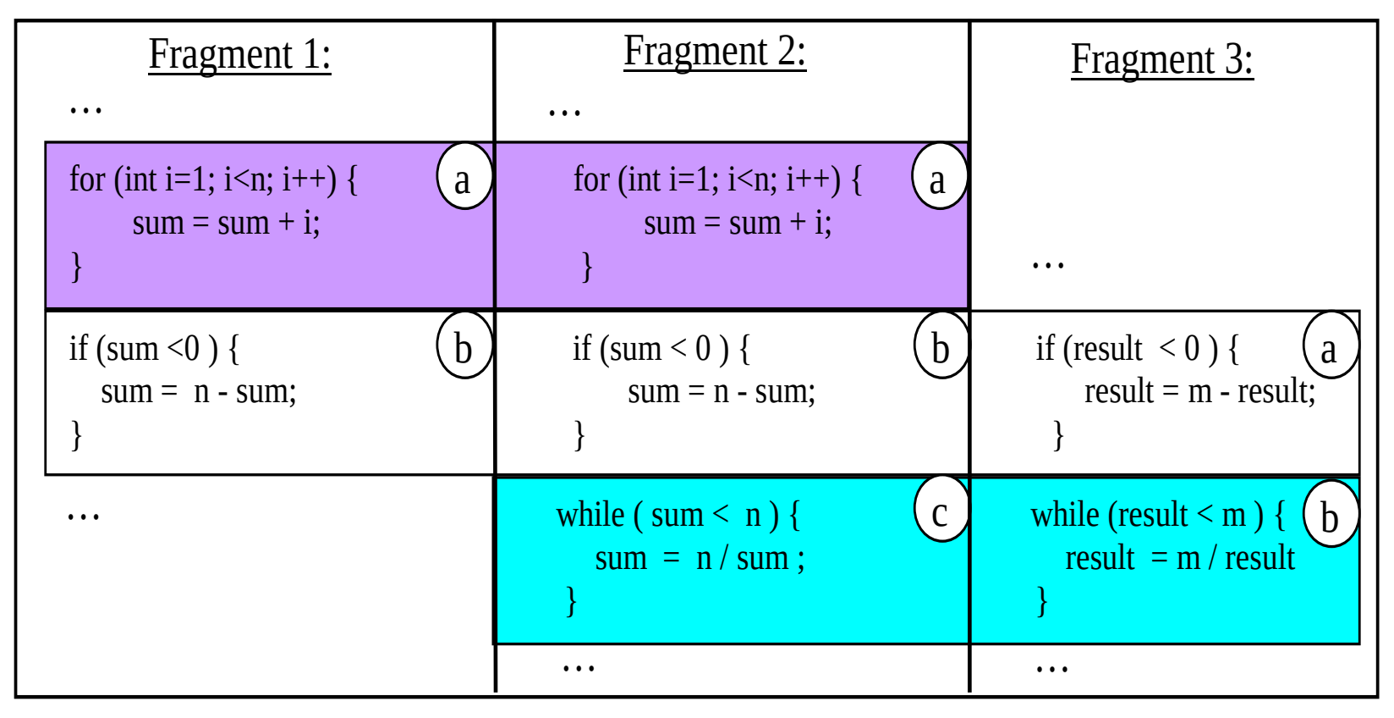
2 Background

Code Fragment: A code fragment (CF) is any sequence of code lines and is identified by its file name and begin-end line numbers, denoted as a triple *(CF.FileName, CF.BeginLine, CF.EndLine)*.

Code Clone: A code fragment *CF2* is a clone of another code fragment *CF1* if they are similar by some given definition of similarity, that is, *f(CF1) = CF2* where *f* is the similarity function (see *clone types* below).

Clone Pairs/Classes: Two fragments that are similar to each other form a *clone pair* (e.g., *(CF1,CF2)*), and when many fragments are similar, they form a *clone class* or *clone group*.

In this example, five clone pairs, <F1(a), F2(a)>, <F1(b), F2(b)>, <F2(b), F3(a)>, <F2(c), F3(b)> and <F1(b), F3(a)>. But actually three clone pairs, <F1(a + b), F2(a + b)>, <F2(b + c), F3(a + b)> and <F1(b), F3(a)>. One clone class is <F1(b), F2(b), F3(a)>.



```
Fragment 1:
...
for (int i=1; i<n; i++) {
      sum = sum + i;
}
if (sum <0 ) {
   sum =  n - sum;
}
...
```

```
Fragment 2:
...
for (int i=1; i<n; i++) {
      sum = sum + i;
}
if (sum < 0 ) {
    sum = n - sum;
}
while ( sum <  n ) {
   sum  =  n / sum ;
}
...
```

```
Fragment 3:
...
if (result  < 0 ) {
     result = m - result;
}
while (result < m ) {
   result  = m / result
}
...
```

Clone Types: The definition of clone is inherently vague in the literature [8]. However, the following four types can roughly be defined [3, 8].

Type 1: Identical code fragments except for variations in whitespace, layout and comments.

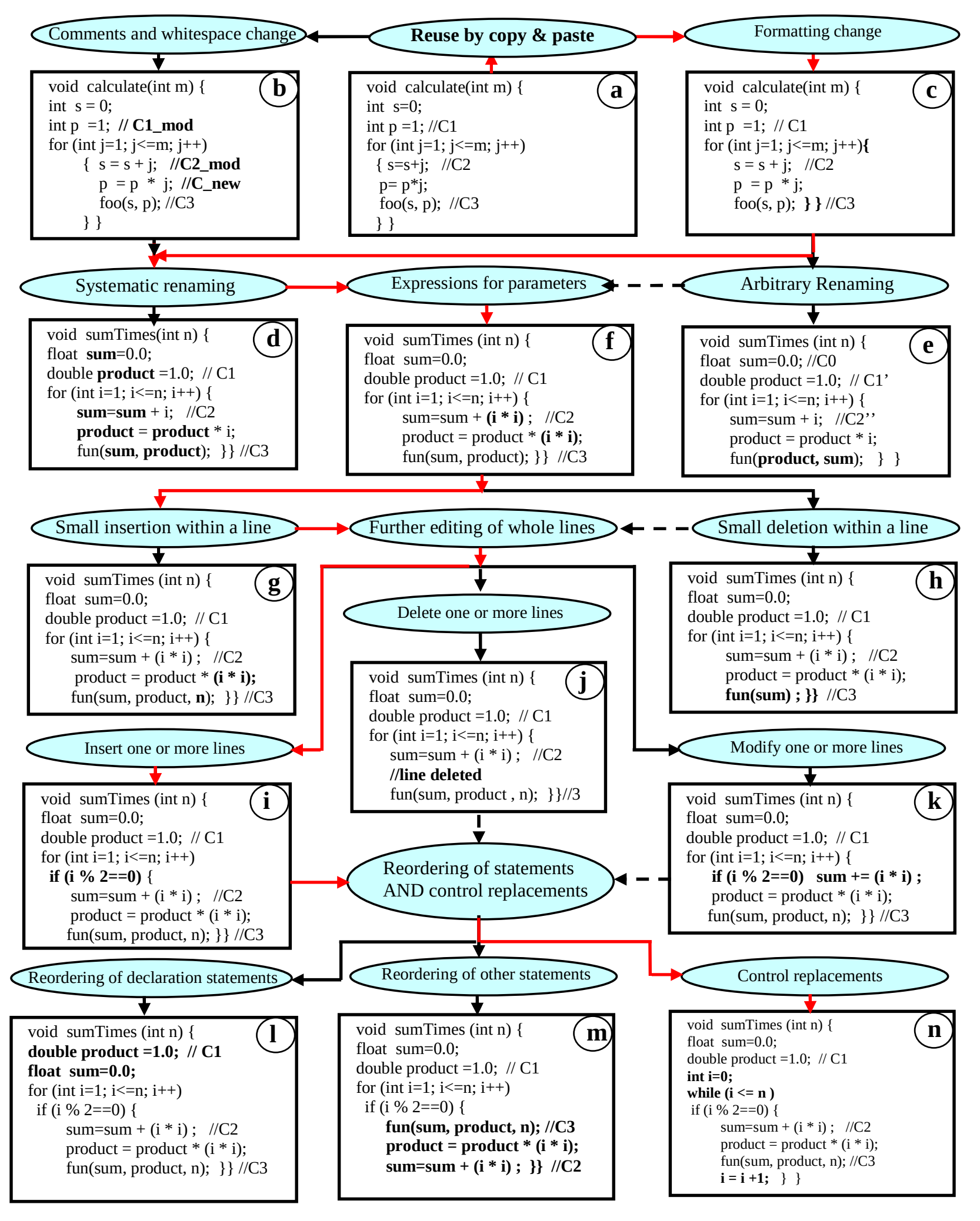
Type 2: Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.

Type 3: Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.

Type 4: Two or more code fragments that perform the same computation but are implemented by different syntactic variants.

3 Editing Taxonomy and Mutation Operators for Cloning

- For any mutation-based analysis, availability of a set of representative mutation operators is a primary concern.
- For example, numerous mutation generators are available for generating potential "bugs" in various languages [1].
- However, mutation operators for code cloning have to our knowledge NOT been studied so far.
- Thus, we have designed an editing taxonomy of different clone types [7] by studying the literature [8]. The taxonomy is also validated by studying the copy/paste patterns of the function clones [5] in our empirical studies [10].
- This taxonomy has been used to design mutation operators for cloning.


Comments and whitespace change
Reuse by copy & paste
Formatting change
Systematic renaming
Expressions for parameters
Arbitrary Renaming
Small insertion within a line
Further editing of whole lines
Small deletion within a line
Delete one or more lines
Insert one or more lines
Modify one or more lines
Reordering of statements AND control replacements
Reordering of declaration statements
Reordering of other statements
Control replacements

```
a
void  calculate(int m) {
int  s=0;
int p =1; //C1
for (int j=1; j<=m; j++)
 { s=s+j;   //C2
  p= p*j;
  foo(s, p);  //C3
 } }
```

```
b
void  calculate(int m) {
int  s = 0;
int p  =1;  // C1_mod
for (int j=1; j<=m; j++)
      {  s = s + j;   //C2_mod
         p  = p  *  j;  //C_new
         foo(s, p); //C3
      } }
```

```
c
void  calculate(int m) {
int  s = 0;
int p  =1;  // C1
for (int j=1; j<=m; j++){
       s = s + j;   //C2
       p  = p  * j;
       foo(s, p);  } } //C3
```

```
d
void  sumTimes(int n) {
float  sum=0.0;
double product =1.0;  // C1
for (int i=1; i<=n; i++) {
      sum=sum + i;   //C2
      product = product * i;
      fun(sum, product);  }} //C3
```

```
e
void  sumTimes (int n) {
float  sum=0.0; //C0
double product =1.0;  // C1'
for (int i=1; i<=n; i++) {
      sum=sum + i;   //C2''
      product = product * i;
      fun(product, sum);   }  }
```

```
f
void  sumTimes (int n) {
float  sum=0.0;
double product =1.0;  // C1
for (int i=1; i<=n; i++) {
        sum=sum + (i * i) ;   //C2
        product = product * (i * i);
        fun(sum, product); }}  //C3
```

```
g
void  sumTimes (int n) {
float  sum=0.0;
double product =1.0;  // C1
for (int i=1; i<=n; i++) {
     sum=sum + (i * i) ;   //C2
      product = product * (i * i);
     fun(sum, product, n);  }} //C3
```

```
h
void  sumTimes (int n) {
float  sum=0.0;
double product =1.0;  // C1
for (int i=1; i<=n; i++) {
       sum=sum + (i * i) ;   //C2
       product = product * (i * i);
       fun(sum) ; }}  //C3
```

```
i
void  sumTimes (int n) {
float  sum=0.0;
double product =1.0;  // C1
for (int i=1; i<=n; i++)
 if (i % 2==0) {
      sum=sum + (i * i) ;   //C2
      product = product * (i * i);
     fun(sum, product, n); }} //C3
```

```
j
void  sumTimes (int n) {
float  sum=0.0;
double product =1.0;  // C1
for (int i=1; i<=n; i++) {
    sum=sum + (i * i) ;   //C2
    //line deleted
    fun(sum, product , n);  }}//3
```

```
k
void  sumTimes (int n) {
float  sum=0.0;
double product =1.0;  // C1
for (int i=1; i<=n; i++) {
      if (i % 2==0)   sum += (i * i) ;
      product = product * (i * i);
     fun(sum, product, n);  }} //C3
```

```
l
void  sumTimes (int n) {
double product =1.0;  // C1
float  sum=0.0;
for (int i=1; i<=n; i++)
  if (i % 2==0) {
        sum=sum + (i * i) ;   //C2
        product = product * (i * i);
        fun(sum, product, n);  }} //C3
```

```
m
void  sumTimes (int n) {
float  sum=0.0;
double product =1.0;  // C1
for (int i=1; i<=n; i++)
  if (i % 2==0) {
      fun(sum, product, n); //C3
      product = product * (i * i);
      sum=sum + (i * i) ;  }}  //C2
```

```
n
void  sumTimes (int n) {
float  sum=0.0;
double product =1.0;  // C1
int i=0;
while (i <= n )
 if (i % 2==0) {
       sum=sum + (i * i) ;   //C2
       product = product * (i * i);
       fun(sum, product, n); //C3
       i = i +1;   }  }
```

In the following table we list the different mutation operators.

Name	Random Editing Activities	Ref**	Type
mCW	Changes in whitespace and	a => b	Type 1
mCC	Changes in comments		
mCF	Changes in formating	a => c	
mSRI	Systematic renaming of identifiers*	c => d	Type 2
mARI	Arbitrary renaming of identifiers*	c => e	
mRPE	Replacement of identifiers with expressions	d => f	
mSIL	Small insertions within a line	f => g	Type 3
mSDL	Small deletions within a line	f => h	
mILs	Insertions of one or more lines	g => i	
mDLs	Deletions of one or more lines	g => j	
mMLs	Modifications of whole line(s)	g => k	
mRDS	Reordering of declarations	i => l	Type 4
mROS	Reordering of other statements+	i => m	
mCR	Replacing one type of control by another	i => n	

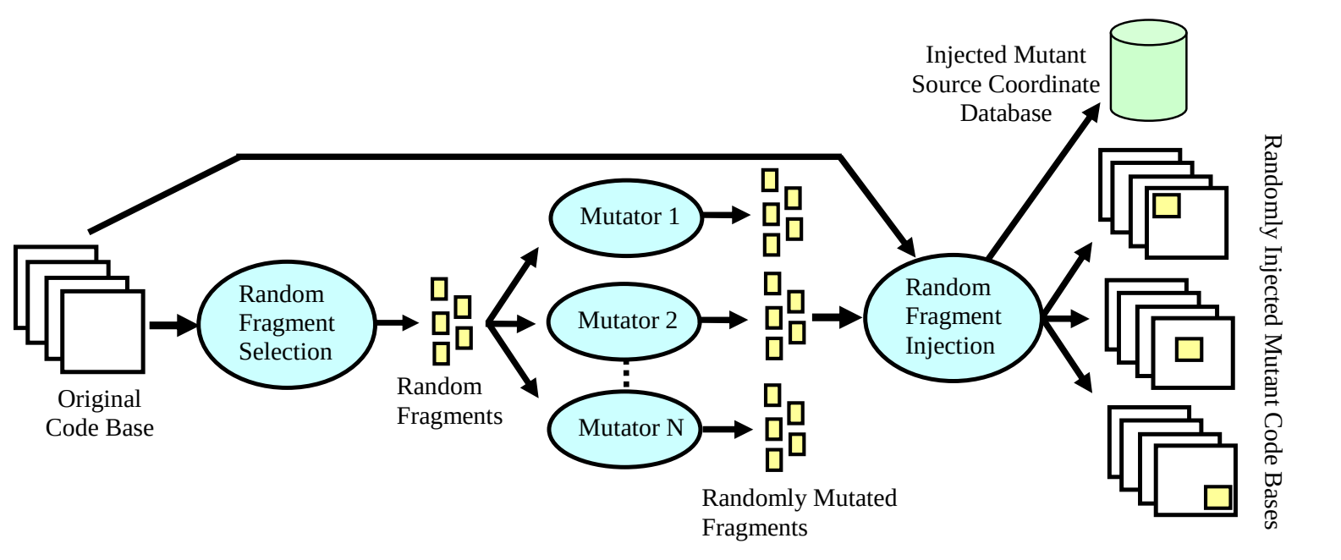
**Refers to the clone taxonomy above
*Function names, variables, data types and literal values
+Data-dependent or independent statements

- Each mutation operator performs single level editing as of the table above.
- Combination of the mutation operators has been used to perform multiple level editing, e.g. (Original Fragment (a) => Formatting change (c) => systematic renaming of identifiers (d) => expressions for parameters (f) => small insertion within a line (g) => insertion of new lines (i) => control replacement (n)) by following the solid (red) lines on the example taxonomy.
- The source transformation language TXL [4] is used to implement the mutation operators.

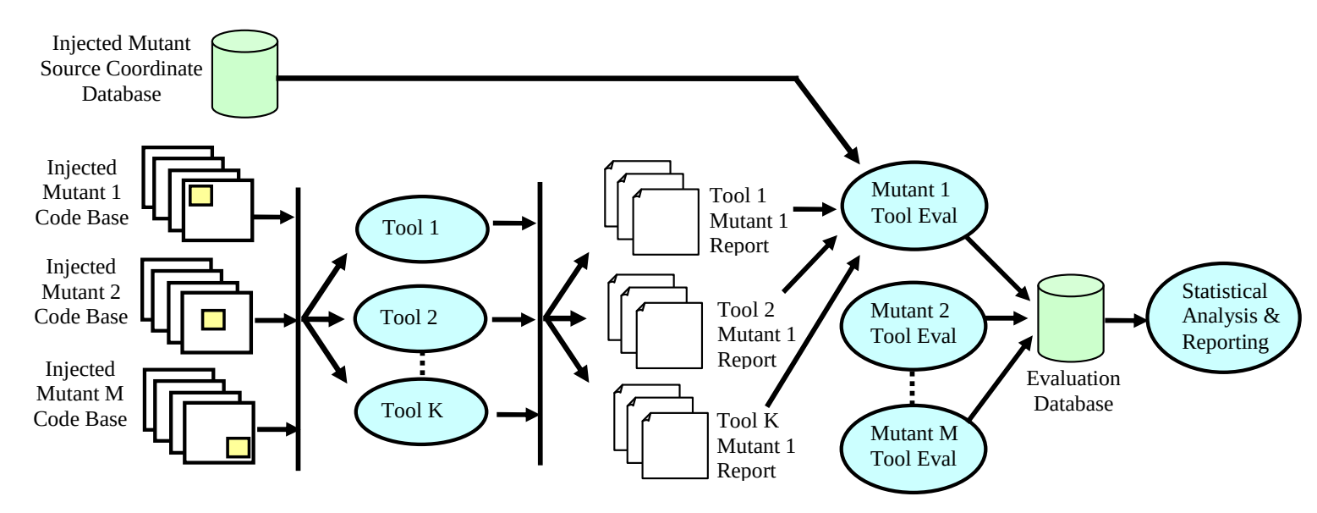
4 The Framework

The framework has two main phases as follows:

Generation Phase: Randomly mutated clone fragments are generated from the original code base and randomly injected into the code base to get mutated code bases.


Original Code Base
Random Fragment Selection
Random Fragments
Mutator 1
Mutator 2
Mutator N
Randomly Mutated Fragments
Random Fragment Injection
Injected Mutant Source Coordinate Database
Randomly Injected Mutant Code Bases

Evaluation Phase: The mutated code bases are used to evaluate and compare clone detection tools. The random mutation and injection steps allow for thousands of randomly placed clones to be generated.



5 Measurement of Recall

Recall definition is the usual one in IR research, that is, the number of items detected divided by the total number of detectable items.

If the mutant clone *moCF* of original code fragment *oCF* injected into mutant code base *mioCB* of code base *oCB* is "killed" (i.e., *(oCF, moCF)* is detected as a clone pair) by the detector, then its recall for that clone is 1, otherwise it is 0. We can denote this decision by:

$$R_T^{(oCF,moCF)} = \begin{cases} 1, & \text{if } \textit{(oCF, moCF)} \text{ is detected by } T \text{ in } \textit{mioCB};\\ 0, & \text{otherwise.} \end{cases}$$

The same mutated code fragment, *moCF* can be randomly injected to the original code base, *oCB* any number of times, producing *n* different mutated / injected versions of *oCB*, say *mioCB1, mioCB2 ... mioCBn*.

The random fragment selector chooses *m* code fragments (say *oCF1, oCF2 ... oCFm*) from the code base, and each of them will be mutated by each mutation operator *dmOP* producing mutated code fragments *moCF1, moCF2 ... moCFm*.

Thus, the recall for mutation operator *dmOP* for tool *T* is given by:

$$R_T^{dmOP} = \frac{\sum_{i=1}^{n*m} R_T^{(oCFi,moCFi)}}{m*n}$$

Similarly, recall of tool *T* for Type 1 clones (three mutation operators and their combinations) can be defined as:

$$R_T^{Typ1} = \frac{\sum_{i=1}^{n*m*(3+4)} R_T^{(oCFi,moCFi)}}{m*n*(3+4)}$$

The overall recall for tool *T* is the summary of recall for the *l* clone mutation operators and *c* combinations applied *n* times to *m* selected code fragments, given by:

$$R_T^{overall} = \frac{\sum_{i=1}^{n*m*(l+c)} R_T^{(oCFi,moCFi)}}{m*n*(l+c)}$$

6 Measurement of Precision

Using the notation of the previous subsection, let us say that for a mutated code fragment *moCF* created by mutation operator *dmOP*, a tool *T* reports *k* clone pairs, *(moCF, CF1), (moCF, CF2) ... (moCF, CFk)* in mutant code base *mioCB*.

If automatic validation reports that *v* of these are valid, then the unit precision of the tool *T* for the single injection of *moCF* for clone type/mutation operator *dmOP* is as follows:

$$P_T^{dmOP\ w.r.t.\ single\ injection\ of\ moCF} = \frac{v}{k}$$

Similarly to the previous section, the precision for mutation operator *dmOP*, for *Type 1* and for a tool *T* (overall precision) is given by:

$$P_T^{dmOP} = \frac{\sum_{i=1}^{n*m} v_i}{\sum_{i=1}^{n*m} k_i}$$

$$P_T^{Typ1} = \frac{\sum_{i=1}^{n*m*(3+4)} v_i}{\sum_{i=1}^{n*m*(3+4)} k_i}$$

$$P_T^{overall} = \frac{\sum_{i=1}^{n*m*(l+c)} v_i}{\sum_{i=1}^{n*m*(l+c)} k_i}$$

7 Mapping of Code Fragments

We say that a code fragment *CF1* is *contained* by another fragment *CF2* if both are in the same file, and and the range of line numbers of *CF1* is within the range of line numbers of *CF2*. In algorithmic form,

```
boolean isContained(CF CF1, CF CF2) {
   return ((CF1.FileName == CF2.FileName)
       AND (CF1.BeginLine >= CF2.BeginLine)
       AND (CF1.EndLine <= CF2.EndLine))
}
```

The following algorithm implements our definition of detection for a mutant pair (MP) consisting of clone mutant *moCF* of original fragment *oCF* and a tool *T*'s clone candidate set (CSet) of detected clone pairs *C*:

```
boolean isDetected(MP (oCF, moCF), CSet C) {
   for each clone pair (CF1, CF2) in C {
       if ((isContained(oCF, CF1)
              AND isContained(moCF, CF2))
            OR (isContained(moCF, CF1)
                 AND isContained(oCF, CF2)))
            return True;
    }
    return False;
}
```

To measure precision, we need to find all pairs in *C* for which one of the fragments is the mutant clone *moCF*:

```
CSet validateUs(CF moCF, CSet C){
   CSet ValidateMe = {};
   for each clone pair (CF1, CF2) in C {
       if (isContained(moCF, CF1)
              OR isContained(moCF, CF2))
           ValidateMe = ValidateMe + (CF1, CF2);
   }
   return ValidateMe;
}
```

8 Validation of Clone Pairs

- Recall measurement is completely automatic and no validation of clone pairs is required.
- But, to accurately measure precision we need to validate those few clone pairs that are associated with the mutant code fragment.
- We develop a clone validator based on our NICAD [6, 10] tool.
- The validator is well aware of the mutation operators applied and changes made on the cloned fragment, and thus can accurately measure their real similarity.

9 Adapting Tools to the Framework

- The tool should be run from the command line (most tools usually do)
- It should provide a textual report of the detected clones, the usual column-oriented textual format of full file name and begin/end line numbers of the code fragments of the candidate clone pairs.

10 Conclusion

- Existing studies for empirically evaluating clone detection tools have had several limitations.
- We have designed a new approach for evaluating clone detection tools in a controlled way by borrowing an established technique from the testing community – mutation-based analysis.
- An experiment was successfully conducted with three different variants of our NICAD tool.
- The framework can run subject tools with varying tunable parameters suitable for identifying different types of clones.
- The framework is language-specific, and currently supports only C, Java and C#. However, since the majority of the framework is language-independent, it is not difficult to add new languages.
- Detailed description of the framework can be found at [11] and an earlier outline at [9].

References

[1] J. H. Andrews, L. C. Briand and Y. Labiche. Is Mutation an Appropriate Tool for Testing Experiments? In *ICSE*, pp. 402-411, 2005.

[2] B.S. Baker. Finding Clones with Dup: Analysis of an Experiment. *IEEE TSE*, Vol. 33(9):608-621, 2007.

[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE TSE*, Vol. 33(9): 577-591, 2007.

[4] J.R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190-210, 2006.

[5] C.K. Roy and J.R. Cordy. WCRE'08 Clone Results: `http://www.cs.queensu.ca/home/stl/download/NICADOutput/`.

[6] C.K. Roy and J.R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *ICPC*, pp. 172-181, 2008.

[7] C.K. Roy and J.R. Cordy. Scenario-Based Comparison of Clone Detection Techniques. In *ICPC*, pp. 153-162, 2008.

[8] C.K. Roy and J.R. Cordy. *A Survey on Software Clone Detection Research.* Queen's School of Computing TR 2007-541, 115 pp., 2007.

[9] C.K. Roy and J.R. Cordy. Towards a Mutation-Based Automatic Framework for Evaluating Clone Detection Tools. In *C3S2E*, Student Poster, pp. 137-140, 2008.

[10] C.K. Roy and J.R. Cordy. An Empirical Study of Function Clones in Open Source Software. In *WCRE*, pp. 81-90, 2008.

[11] C.K. Roy and J.R. Cordy. A Mutation / Injection-based Automatic Framework for Evaluating Code Clone Detection Tools. Submitted to *ICSE*, 11 pp., 2009.