NICAD: A Next Generation Clone Detection Tool

Chanchal K. Roy and James R. Cordy

Software Technology Laboratory, School of Computing
Queen's University, Kingston, Ontario, Canada

1. Introduction

- Intentional copy/paste a common reuse technique in software development*
- Previous studies report 7% - 23% cloned code in various kinds of software systems, Baker WCRE'95

In response, many clone detection methods

- Lightweight text-based and lexical
  - High recall and text accuracy
  - But results aren't meaningful syntactic units
- Heavier parser-based techniques
  - Meaningful units and high precision
  - But expensive comparison and low recall
- Neither handles near-miss clones well

Our plan, a hybrid:

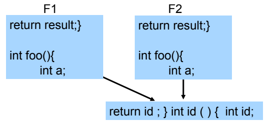
- Combines strengths, overcomes limitations of both text-based and AST-based techniques
  - Proven effective (with high precision and recall) in finding near-miss function clones
- A hybrid parser / text line-based technique
  - And other novel features of other approaches

2. Overview of Existing Methods

Bellon et al. TSE'07, Roy and Cordy ICPC'08, SCP'09, TechReport'07

Approach	Strengths	Limitations
Text-Based	100% Precision	Sensitive to formatting & editing, Non-syntactic clones
Token-Based	Fast, High recall, Normalization	Medium precision, Often not syntactic clones
Tree-Based	Syntactic clones, High precision	Low recall, Fully-fledged parser, Expensive tree comparison
Metrics-Based	Fast, Syntactic clones	Medium precision and recall, Fully-fledged parser
Graph-Based	Might detect semantic clones	Low recall, Not scaled, Expensive graph comparison

3. Text- and Token-Based Often Detect Non-Syntactic Clones



F1

```
return result;}

int foo(){
      int a;
```

F2

```
return result;}

int foo(){
      int a;
```

```
return id ; } int id ( ) {  int id;
```

We Do: Structural Extraction

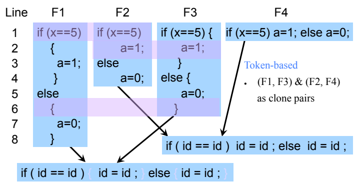
- Use robust island grammars to isolate and extract
  - Meaningful units for comparison
  - Example: begin-end block, function block or any structured block
  - Source coordinate of the units
- No need of fully-fledged parser
  - Standalone, only TXL grammar

4. Text-Based: Sensitive to Formatting Changes

Line	F1	F2	F3	F4
1	if (x==5)	if (x==5)	if (x==5)	if (x==5) a=1; else a=0;
2	{	a=1;	{	
3	a=1;	else	a=1;	
4	}	a=0;	}	
5	else		else {	
6	{		a=0;	
7	a=0;		}	
8	}			

- Text line-based
  - Not clones

Token-Based: Not Fully Robust to Formatting Changes



Line	F1	F2	F3	F4
1	if (x==5)	if (x==5)	if (x==5) {	if (x==5) a=1; else a=0;
2	{	a=1;	a=1;	
3	a=1;	else	}	
4	}	a=0;	else {	
5	else		a=0;	
6	{		}	
7	a=0;			
8	}			

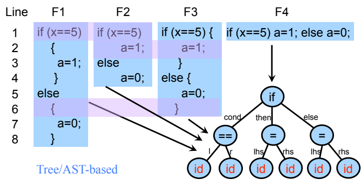
```
if ( id == id )  id = id ; else  id = id ;
```

```
if ( id == id ) {  id = id ; }  else {  id = id ; }
```

Token-based

- (F1, F3) & (F2, F4) as clone pairs

Tree-Based: Robust to Formatting Changes



Line	F1	F2	F3	F4
1	if (x==5)	if (x==5)	if (x==5) {	if (x==5) a=1; else a=0;
2	{	a=1;	a=1;	
3	a=1;	else	}	
4	}	a=0;	else {	
5	else		a=0;	
6	{		}	
7	a=0;			
8	}			

Tree/AST-based

- Find as clones, but expensive , low recall, not good for finding near-miss clones

We Do: Standard Pretty-Printing

- Parser eliminates commenting (if any) and input formatting differences
  - Pretty-printing to preserve tree structure in text

```
redefine if_statement
  if ( [expr] ) {      [IN][NL]
     [statement]       [EX]
  }
  [opt else_statement]
end redefine
```

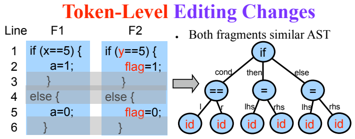
```
redefine else_statement
  else  {             [IN][NL]
     [statement]       [EX]
  }
end redefine
```

Standard Pretty-Printing

Line	F1	F2	F3	F4
1	if (x==5) {	if (x==5) {	if (x==5) {	if (x==5) {
2	a=1;	a=1;	a=1;	a=1;
3	}	}	}	}
4	else {	else {	else {	else {
5	a=0;	a=0;	a=0;	a=0;
6	}	}	}	}

- Even "{" and "}" are added to fragments F2 and F3
- Text-line comparison now finds them exactly similar
- Form a clone class, {F1, F2, F3, F4} as of tree-based method but avoids expensive tree comparison
- Because of text-comparison, precision is now 100%

5. Token- and Tree-Based: Robust to Token-Level Editing Changes



Line	F1	F2
1	if (x==5) {	if (y==5) {
2	a=1;	flag=1;
3	}	}
4	else {	else {
5	a=0;	flag=0;
6	}	}

- Both fragments similar AST

- Text-based are sensitive to any changes
- Token-based methods give lots of false positives, Bellon et al. TSE'07
- Tree-based methods are expensive, low recall, and not as high precision as of text-based methods

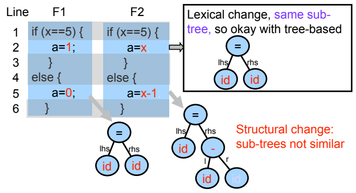
We do: Flexible Token-Normalization

Line	F1	F2
1	if (id==5) {	if (id==5) {
2	id=1;	id=1;
3	}	}
4	else {	else {
5	id=0;	id=0;
6	}	}

- Text-line comparison now finds them as clone pair
- High flexibility in token-normalization

- We use parser, but avoid the problems of tree-based methods by text-comparison
- As we first extract structural units for comparison, we avoid problems of classical text-based and token-based methods

Text-, Token- and Tree-Based are Sensitive to Structural Changes



Line	F1	F2
1	if (x==5) {	if (x==5) {
2	a=1;	a=x
3	}	}
4	else {	else {
5	a=0;	a=x-1
6	}	}

Lexical change, same sub-tree, so okay with tree-based

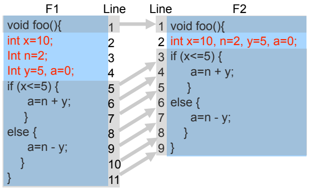
Structural change: sub-trees not similar

We Do: Flexible TXL Rules for Structural Normalization

Line	F1	F2
1	if (x==5) {	if (x==5) {
2	a=id;	a=id
3	}	}
4	else {	else {
5	a=id;	a=id
6	}	}

- Text-line comparison finds them
- High Flexibility in Abstraction
- LHS of the assignments Not normalized to avoid false positives.
- Avoid the problems of tree-based

6. Flexible Code Filtering



F1

Line	F1
1	void foo(){
2	int x=10;
3	Int n=2;
4	Int y=5, a=0;
5	if (x<=5) {
6	a=n + y;
7	}
8	else {
9	a=n - y;
10	}
11	}

F2

Line	F2
1	void foo(){
2	int x=10, n=2, y=5, a=0;
3	if (x<=5) {
4	a=n + y;
5	}
6	else {
7	a=n - y;
8	}
9	}

- After removing the declaration statements, text-line comparison will find them as clone pair with high accuracy

7. Flexible Pretty-printing

- Example, "for" headers

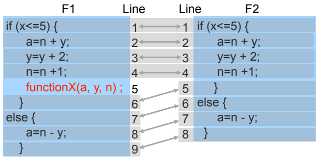
Line	F1	F2	F3
1	for(i=0;i<10;i++)	for(i=1;i<10;i++)	for(j=2;j<100;j++)

0% same on text-line comparison

Line	F1	F2	F3
1	for(	for(	for(
2	i=0;	i=1;	j=2;
3	i<10;	i<10;	j<100;
4	i++)	i++)	j++)

{F1, F2} 75% same
{F1, F3} 25% Same
{F3, F3} 25% Same

8. Text-Line Comparison with Gaps



F1	Line	Line	F2
if (x<=5) {	1	1	if (x<=5) {
a=n + y;	2	2	a=n + y;
y=y + 2;	3	3	y=y + 2;
n=n +1;	4	4	n=n +1;
functionX(a, y, n) ;	5	5	}
}	6	6	else {
else {	7	7	a=n - y;
a=n - y;	8	8	}
}	9		

LCS: 1-2-3-4-6-7-8-9 (w.r.t. F1)

$$UPI\_F1 = \frac{\text{No. of unique items/lines in F1 w.r.t. F2}}{\#Lines(F1)} \text{ x } 100$$

$$= \frac{\#Lines(F1) - \#Lines(LCS)}{\#Lines(F1)} \text{ x } 100$$

Similarly, for fragment F2,

$$UPI\_F2 = \frac{\#Lines(F2) - \#Lines(LCS)}{\#Lines(F2)} \text{ x } 100$$

- **Definition of Clone**
  - Given a UPI threshold UPI_T, fragments F1 and F2 form a clone pair if and only if,

  (UPI_F1 <= UPI_T) AND (UPI_F2 <= UPI_T)

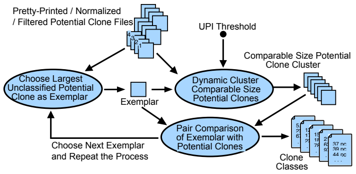
- E.g., if UPI_T is 20%, then two fragments considered clones if 80% of pretty-printed text lines identical.

For the running example, #Lines(LCS)=8

UPI_F1=11% and UPI_F2=0%
If UPI_T==10%, not clone pair
If UPI_T==15%, {F1, F2} clone pair

9. Comparing the Potential Clones

- LCS algorithm compares two extracted units /potential clones at a time
  - In principle, must compare every pair of potential clones => quadratic w.r.t. no. of potential clones
- Three major strategies to improve
  - Apply dynamic clustering based on the size of a chosen exemplar and the UPI threshold
  - Farm out pair comparisons to multiple processors
  - Make comparisons one-pass using exemplars

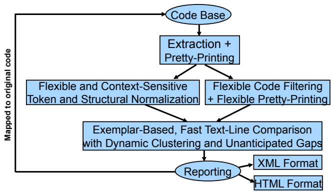


Pretty-Printed / Normalized / Filtered Potential Clone Files
4
3
2
1
UPI Threshold
Comparable Size Potential Clone Cluster
Choose Largest Unclassified Potential Clone as Exemplar
Exemplar
Dynamic Cluster Comparable Size Potential Clones
Pair Comparison of Exemplar with Potential Clones
Choose Next Exemplar and Repeat the Process
Clone Classes
37 pc
39 pc
44 pc

10. Reporting/Output Generation

- Two forms of output
  - XML database of clone classes with source coordinate information (file name, begin-end line numbers)
    - Suitable for use by IDEs, statistical analysis / reporting tools
  - HTML website report of clone classes
- Original raw source code reported
  - Using source coordinate annotations from potential clones

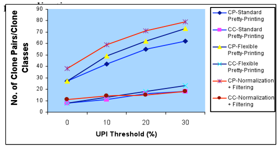
11. Conceptual Diagram of NICAD



Code Base
Extraction + Pretty-Printing
Flexible and Context-Sensitive Token and Structural Normalization
Flexible Code Filtering + Flexible Pretty-Printing
Exemplar-Based, Fast Text-Line Comparison with Dynamic Clustering and Unanticipated Gaps
Reporting
XML Format
HTML Format
Mapped to original code

12. First Experimental Results

- Weltab
  - Studied effect of flexible pretty-printing,



13. Large Empirical Studies

- Comprehensive in-depth evaluation of clone properties -Roy and Cordy WCRE'08
  - In different dimensions
  - Three different languages (10 C, 7 Java and 7 C#)
  - Diverse varieties of applications
  - All open source systems including complete Linux Kernel
  - 4 KLOC- 6.3 MLOC
  - In varying UPI thresholds
- Also evaluated with a mutation / injection based evaluation framework, Roy and Cordy, Mutation'09
  - NICAD was found very good both for precision and recall for different types of fine-grained clones

References

- 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.
- C.K. Roy and J.R. Cordy. An Empirical Study of Function Clones in Open Source Software. In *WCRE* , pp. 81-90, 2008 (Invited for special issue).
- C.K. Roy and J.R. Cordy. Near-miss Function Clones in Open Source Software: An Empirical Study. In JSME , 23 pp., 2009 (submitted).