NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization

Chanchal K. Roy and James R. Cordy
School of Computing, Queen's University
Kingston, ON, Canada K7L 3N6
{croy, cordy}@cs.queensu.ca

Abstract

This paper examines the effectiveness of a new language-specific parser-based but lightweight clone detection approach. Exploiting a novel application of a source transformation system, the method accurately finds near-miss clones using an efficient text line comparison technique. The transformation system assists the method in three ways. First, using agile parsing it provides user-specified flexible pretty- printing to remove noise, standardize formatting and break program statements into parts such that potential changes can be detected as simple linewise text differences. Second, it provides efficient flexible extraction of potential clones to be compared using island grammars and agile parsing to select granularities and enumerate potential clones. Third, using transformation rules it provides flexible code normalization to allow for local editing differences between similar code segments and filtering out of uninteresting parts of potential clones. In this paper we introduce the theory and practice of the framework and demonstrate its use in finding function clones in C code. Early experiments indicate that the method is capable of finding near-miss clones with high precision and recall, and with reasonable performance.

1. Introduction

Copying a code fragment and reusing it by pasting with or without minor modifications is a common practice in software development environments. As a result software systems often have sections of code that are similar, called software clones or code clones. Previous research shows that a significant amount of code (between 7% to 23%) of a software system is cloned code [3, 5, 22, 26]. While programmers often practise cloning with clear intent [23] and it is beneficial in certain situations [21], one of the major difficulties with such duplicated fragments is that if a bug is detected in a code fragment, all the fragments similar to it should be investigated to check for same bug [25]. Moreover, when enhancing or adapting a piece of code, duplicated fragments can multiply the work to be done [19].

From a program comprehension point of view, clones carry important domain knowledge and thus studying the clones in a system can assist in understanding it [19]. Moreover, by refactoring the clones detected, one can potentially improve understandability, maintainability and extensibility, and reduce the complexity of the system [15].

Fortunately, several (semi-)automated techniques for detecting code clones have been proposed (c.f., Section 11). Several studies show that lightweight text-based techniques can find clones with high accuracy and confidence, but detected clones often do not correspond to appropriate syntactic units [7, 30]. Parser-based syntactic (AST-based) techniques, on the other hand, find syntactically meaningful clones but tend to be more heavyweight, requiring a full parser and subtree comparison method. Moreover, neither text-based nor parser-based techniques have been found to be effective in detecting near-miss clones [7].

In this paper, we propose a multi-pass approach which is parser-based and language-specific but reasonably lightweight, using simple text line rather than subtree comparison to achieve good time and space complexity. We exploit the benefits of TXL [9] to efficiently identify and extract potential syntactic clones with pretty-printing to eliminate formatting differences and noise. TXL's agile parsing [11] allows us to flexibly select granularity, and to tune the pretty-printing of potential clones to introduce additional line breaks such that potential variances within statements and other structures can be accurately reflected using a simple text line comparison. TXL's transformation rules allow us to add flexible code normalization and filtering of uninteresting or irrelevant sections in the potential clones, yielding accurate minimal differences that are easily traced back to original source using source coordinates.

Our approach is lightweight in the sense that, like other text-based techniques (e.g., *Duploc* [13]), we work directly on program source text. Although pretty-printing, code normalization and filtering all use TXL's agile parsing and

transformation rules, they can be done on a source file-by-file basis and are scalably independent of the program's overall structure. The method can be applied to any language for which we have an approximate (pretty-printing) TXL grammar and (optionally) examples of the desired variances and code normalizations for the language. It is language-specific in that sense.

Although straightforward in TXL (for example, see [8]), in this work we are not aiming at detecting renamed or consistently renamed clones, where identifiers and literals may be changed in a copied fragment. We assume that in the case of intentional clones the user normally does not arbitrarily perform renaming operations on the reused code unless otherwise necessary. (However, Li et al. [25] have reported that there may actually be a significant number of renamed clones, and certainly we will attack this problem in future work.) There are also well established techniques to detect such parameterized clones (e.g., *Dup* [3]) and renamed clones (e.g., *CCFinder* [20]). Thus we are primarily aiming at near-miss clones resulting from intentionally copied fragments that may have undergone editing to adapt to the new context (see [29] for a range of examples).

The rest of the paper is organized as follows. Following a short introduction to TXL, agile parsing and island grammars in Section 2, we provide an overall summary of our approach in Section 3. In Section 4 we describe our flexible multi-granular method for extraction of potential clones using TXL. In Section 5 we discuss adapting pretty-printing to eliminate noise, standardize formatting and isolate variance to lines using agile parsing, and in Sections 6 and 7 we add TXL rules to allow for flexible code abstraction and filtering of potential clones respectively. In Section 8 we discuss optimizing the finding of near-miss clones using simple text-line comparison of potential clones, and generating output in Section 9. Section 10 reports our first empirical results in using the framework to detect known function clones in two C programs. Finally, Section 11 discusses the relation of our work with previous techniques, and Section 12 concludes the paper with our next steps.

2. Background

Our approach is based on lightweight agile parsing techniques supported by the TXL source transformation system. TXL [9] is a special-purpose programming language designed to provide rule-based source transformation using functional specification and interpretation. TXL programs have two main parts: a context-free grammar that describes the syntactic structure of inputs to be transformed, and a set of context-sensitive, example-like transformation rules organized in functional programming style.

TXL operates in three phases: *parse*, *transform*, and *unparse*. The parsing phase creates an internal representation of the input as a parse tree under control of a context-free grammar like the one for C *if- then-else* statement below. TXL grammars specify not only input forms for parsing, but also output pretty-printing for unparsing using the special markers [IN] (indent), [EX] (exdent) and [NL] (newline).

```
define if_statement
     'if ( [expr] )       [IN][NL]
         [statement]      [EX]
     [opt else_statement]
end define

define else_statement
     'else               [IN][NL]
         [statement]     [EX]
end define
```

The transformation phase transforms the parse trees created by the parser under control of a set of example-like transformation rules that easily express normalization and abstraction for clone detection, for example to anonymize *if* conditions or normalize identifiers. Finally, the TXL unparsing phase unparses the transformed parse tree to text output with standard spacing and pretty-printing under control of the grammar, and ignoring all input formatting. Commenting and spacing in the input are ignored by default (although it can be preserved if desired).

TXL supports agile parsing [11], which allows nonterminal definitions to be modified by *grammar overrides*, allowing programs to easily specify different interpretations of syntax and different pretty-printing in different programs based on the same grammar. For example, we override a program using the C grammar with the *if-then-else* definition above to modify the pretty-printing to eliminate indenting and newlines in the output:

```
redefine if_statement
     'if ( [expr] )  [statement]  [opt else_statement]
end redefine

redefine else_statement
     'else  [statement]
end redefine
```

Island grammars [12] are a grammar-based method for separating interesting parts of a program (features we are interested in) from uninteresting parts (other features, which need not be precisely parsed). In our context, island grammars provide a simple mechanism to identify the interesting elements to be compared as potential clones. Island grammars also provide robustness [27] by allowing us to use semi-parisng for the language (or dialect) of interest. Island grammars can be coded in TXL either directly in the language grammar or using grammar overrides to specify a dialect of the language in which the islands are embedded.

3. Proposed Approach

This work is an adaptation and significant extension of our previous work on detecting near-miss clones in HTML documents [10], which made use of a robust island grammar to identify and isolate syntactic constructs such as HTML

tables and forms as potential clones. The grammar separated constructs interesting as potential clones (the islands) from the unparsed rest of the HTML code (the water), extracting the islands as a directory of potential clone files to be compared as text lines using the Unix *diff* utility.

In this work, we further explore and extend the basic idea of the approach in the context of clone detection for systems written in C. It differs from our previous work in a general framework using flexible pretty-printing, code normalization and filtering, with fundamental improvements in the comparison algorithm, clustering of potential clones and output generation.

Figure 1 represents a conceptual diagram of our new clone detection process. We call our prototype *NICAD*, a loose acronym for *Accurate Detection of Near-miss Intentional Clones*. The main distinguishing characteristics of our method are the identification and extraction of the set of potential clones, the flexible pretty-printing, normalization and filtering of the potential clone set, the clustering of potential clones to minimize comparison cost, and the reporting of results in terms of original source. In the following sections we provide a detailed description of each component of this new process.

4. Extraction of Potential Clones

Every clone extraction tool designates – sometimes implicitly – the notion of a "minimal clone", the smallest piece of code that the tool considers to be worthwhile to examine on its own. This step is important for two reasons: it reduces the amount of work the clone detector has to do, and makes the results more usable and relevant. The amount of work is cut because the tool does not spend time looking for clones of program entities that are too small, and the results are improved because they are not polluted with information about the "cloning" of single tokens or (small groups of) statements.

As in our previous approach, the clone extractor is responsible for enforcing these minimal clone restrictions. Its task is to extract potential structural clones from the source code for further study, and it is responsible for extracting features no smaller than our designated minimal clone.

The definition of a minimal clone can vary from language to language and application to application. In the case of C systems, we often choose individual structured blocks that are at least 6 lines of source code as minimal clones. However, one could choose any level of structural granularity. For example, we could choose only the whole functions of a system, or the structured statements, or *begin-end* blocks of a certain minimum size.

In our system, we exploit agile parsing and the TXL *extract* function to enumerate our potential clones. The extract function, denoted `[^ ]` in TXL, automatically extracts a set of all embedded instances of one grammatical type (e.g., *statement*), given an instance of another (e.g., *function definition*). Using grammar overrides, we modify the grammar to capture our minimal clones in special nonterminals which can be extracted using a single invocation of the *extract* function. Each potential clone is extracted only once, but if the potential clone we are interested in is nested, the inner candidate is listed twice: once inside its parent and once on its own. All extracted potential clones are stored as text files, annotated with the original source file names and beginning and ending source line numbers of their origin.

During extraction, potential clone files are also automatically stripped of formatting and comments and pretty-printed by TXL according to the grammar's formatting cues. Pretty-printing ensures consistent layout and spacing of code for later comparisons. When code is cloned, it is often changed – whitespace and comments are inserted or removed, block markers are moved around to suit the developer's taste, and so on. Whitespace and comment removal might address these changes to some degree, but does not necessarily eliminate them. Standard pretty-printing guarantees that all code has uniform layout and line breaks, yielding an improvement in comparison accuracy.

5. Flexible Pretty-Printing

In addition to removing the formatting and layout differences between code segments using standard pretty-printing, we can also exploit TXL's agile parsing to introduce flexible pretty-printing specific to clone detection. This special pretty-printing helps us to break different parts of a statement into several lines so that local changes to the parts of a statement can be isolated using a simple line-comparison. Unlike token-based techniques (e.g., *CCFinder*) where each token is an item of the token-sequence, we allow different parts of a statement to be compared at different granularities, appropriate to the particular language structure. An item to be compared in our method may contain one token, or several tokens, according to the particular pretty-printing rules we choose. Because we use a fast text-line-based technique for comparing potential clones to each other, pretty-printing to spread the code over more lines increases the granularity and allows us to choose the appropriate granularity for each particular language and context.

To see the effectiveness of such feature-specific pretty-printing, let us consider three code fragments, each only with only a single fragment (a *for* loop header).

```
Segment 1:  for (i=0; i<10; i++)
Segment 2:  for (i=1; i<10; i++)
Segment 3:  for (j=2; j<100; j++)
```

With a typical line-based technique (e.g.,*Duploc* [13]) with no normalization/transformation option selected, all the three segments are different. With a classical token-based technique (e.g., *CCFinder* [20]) with identifier normalizations, all the segments will be similar and returned

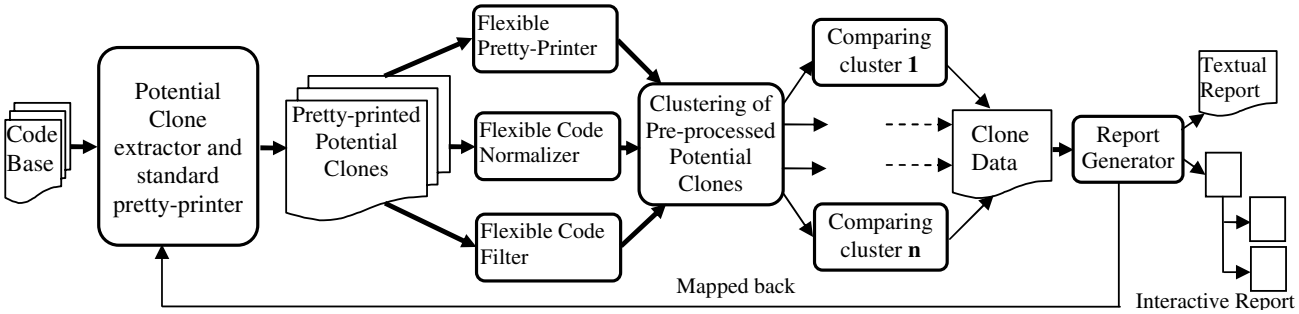

Figure 1. Proposed Clone Detection Process

Table 1. Pretty-printing and Comparing

Line No.	Segment1	Segment2	Segment3	Comparison		
				S1&S2	S1&S3	S2&S3
1	for (	for (	for (	1	1	1
2	i = 0;	i = 1;	j = 2;	0	0	0
3	i < 10;	i < 10;	j < 100;	1	0	0
4	i++)	i++)	j++)	1	0	0
Total Matches				3	1	1
Total Mismatches				1	3	3

as clones. With a classical AST-based technique (e.g., *CloneDR* [5]) where a hashing is used to ignore the leaves of the trees, these three segments will be still similar and will be returned as clones. In fact, using the general definition of clones where only syntactic similarity is considered (e.g., the Type II clones of [7]), these are indeed clones and form a clone class.

In this study, we are primarily interested in accurately finding copied fragments edited following pasting. We anticipate that if a code fragment is intentionally copied for reuse (if the purpose is not plagiarism), then the chances of consistent renaming, or renaming only identifiers and literals, are low (although of course it is also interesting to find such clones [25] and we plan to do so in our future work). Thus, in our study, while *segment 1* and *segment 2* form a clone pair (only the initialization value of i changes), *segment 3* probably does not form any clone pairs with *segment 1* or *segment 2*, because there are several changes, including the variable name.

With a careful pretty-printing (and an appropriate similarity threshold) for the code segments, we can more clearly see that while *segment 1* and *segment 2* are clones, *segment 3* is not. Table 1 shows a comparison where our pretty-printing has been used to break all *for* headers into into four separate lines. Using a naive line-by-line text comparison of the segments and a reasonable similarity threshold, we can accurately determine the similarity between the segments. If we now consider a size-sensitive similarity threshold (see Section 8) of 70%, we can see that only *segment 1* and *segment 2* (with similarity 75%) form a clone pair. Both *segment 1* and *segment 3*, and *segment 2* and *segment 3* are only 25% similar, and thus cannot form clone pairs. We could further break down the code segments such that each token is formatted to a separate line and then catch more changes in the code segments. However, in that case, we would have to reduce the similarity threshold, and in the same time, time complexity would increase.

In TXL specifying such precise flexible pretty-printing is simple. One needs only to override the grammar definition of the intended statement. For example, consider the following TXL grammar definition for *for* headers in C.

```
define for_head
    'for ([opt expr]'; [opt expr]'; [opt expr])
end define
```

Using grammar overrides we can redefine the grammar to break *for* headers into four parts, each on a different line (as in Table 1), simply by adding "$[NL]$" (new line) formatting cues to the grammar for each part as follows.

```
redefine for_head
     'for(          [NL]
     [opt expr] ';  [NL]
     [opt expr] ';  [NL]
     [opt expr] )   [NL]
     [statement]
end redefine
```

Let us consider another example, a function call, say *foo(len, sum)*. A typical approach would either normalize this call as $id()$, or as $id(id, id)$ and then apply an exact matching algorithm. In the first case, all function calls in the code are treated as the same and thus clone detection may generate many false positives. In the second case, all function calls that have two parameters will match, again generating many false matches and in the same time may miss some potential matches (e.g., overloaded function calls in C++ where only number of parameters is changed).

Using flexible pretty-printing, we can stay in the middle. For example, we may consider that for a function call, at least the function name or the number of parameters (and possibly names of the parameters) should be same to have a match with other function calls. We can do this using the same pretty-printing technique, using TXL formatting to break function calls into two lines, the function name,

foo in one line and its parameters (len, sum) on another. If there is another function call in another segment, say *foo(len, sum, product)* (function overloading), it will similarly be broken into two lines by the pretty-printer. Using a naive text-line-based comparison of the two pretty-printed segments, we can see that these two function calls are neither exactly similar nor completely dissimilar. They are in fact in the middle, neither similar (as in normalizing both to $id()$) nor dissimilar (as in normalizing the first call to $id(id, id)$ and the second to $id(id, id, id)$). Using size-sensitive similarity, they are 50% similar. In the same way, we can assign similarity values to each of the statements in the two code segments and more accurately assess the possibility that they are clones.

In the prototype implementation of our method for C, the user can either select from a set of flags controlling the breaking level for each of the statements on how the pretty-printing should be performed, or can change the TXL grammar overrides by hand to achieve other custom granularities.

6. Flexible Code Normalization

While flexible pretty-printing helps us to find near-miss clones, it may require us to adjust the similarity threshold as well (e.g., the more fine-grained the break-up of a statement, the higher the dissimilarity threshold might be). In addition, as with all other methods, it may not be possible to predict in advance where possible changes might have been made in a cloned fragment. Using TXL transformation rules in addition to its parsing and pretty- printing capabilities, we can easily normalize parts of a statement (or whole statements of a given type) to ignore editing differences. By adding normalization to our pretty-printing, we can detect near-miss clones even with a 100% similarity threshold.

Unlike other classical token-based methods, our flexible normalization is not simply limited to global replacement, for example of all identifiers and literals, or simple abstraction, for example of loop bodies. Using TXL rules we can choose to normalize only certain parts of a statement or only statements of a certain type, for example, only *if* statements. Thus we can choose to normalize only those parts that we expect to vary. Using TXL patterns, we can also provide flexibility in applying the normalization, for example choosing only to normalize within a certain type of statement or within a certain level of nesting. In this way, we can be sure of the locations of potential changes in a detected cloned fragment when 100% similarity is used in the comparison phase.

We can also apply both flexible pretty-printing and flexible code normalization together in combination with a dissimilarity threshold (e.g., 90% similarity). This allows us to find near-miss clones that may have changed not only in the normalized places but also some other arbitrary parts that we could not anticipate in code normalization. To see effectiveness of such flexible code normalization, let us consider the case of normalizing conditional statements in C.

Table 2. Typical normalization of an if-then-else

Original Statement	Typical Normalization

`if (x < (n + y))`
`  m = (y + c)-x;`
`else`
`  m = y;`
	`if (id < (id + id))`
	`  id = (id + id)-id;`
	`else`
	`  id = id;`

TXL rules allow us to easily customize and localize normalizations by language feature and context. As an example, we can choose to custom normalize conditional statements. In general there are several conditional statements in a programming language, and using TXL rules we can target only certain ones, and normalize each of them in the most appropriate way. For example, in the case of the C *if-then-else* statement, we can choose just to normalize the *control* part of the statement.

Consider the C *if-then-else* statement of Table 2 (left column). A typical token-based approach will apply global lexical normalization and produce something like the code shown in the right column of Table 2. As general normalization is applied to the entire source, such an approach can produce many false positives [7]. In our method, we apply neither general lexical normalization nor exact string matching. As with flexible pretty-printing, we stay in the middle. For a conditional statement, we can first focus on the control part, keeping the other parts unchanged (or left to other custom normalizations).

In our example, we could simply normalize the control part, $(x < (n + y))$ to $AnyControl$. Clones obtained using such a normalization will have the possibility of having only the control parts edited. However, it may also generate false positives as it normalizes the entire control part. So instead, we could apply the traditional normalization, but only on the control part. For example, $(x < (n + y))$ could be normalized to $(id < (id + id))$ and keep the other parts of the *if-then-else* statement unchanged. (In TXL, this can be implemented using simply the scoped application of a transformation subrule instead of a direct change.) As only the control part is normalized and other parts are unchanged, there will likely be fewer false positives than the general normalization. However, since all the identifiers of the control part are normalized, this could still return false positives.

So we can consider an even more restricted normalization. We can apply normalization only on the right hand side of the control expression, for example to normalize $(x < (n + y))$ to $(x < (id + id))$, keeping the left part unchanged. This will allow identifier changes on the major part of the control, and at the same time avoid false positives as the left-part is unchanged. However, all the above

normalizations can miss clones that have been structurally modified in the control part only (e.g., $(x < (n + y))$ could be changed to $(x < (n*y))$).

In the end we might refine to a more general normalization. For example, we can normalize $(x < (n * y))$ to $(x < rightControl)$. While this will allow more structural changes, it will avoid false positives in two ways. First, its left part is unchanged and second, other parts of the *if-then-else* statement are either unchanged or normalized by other applicable options. Moreover, clones obtained with this normalization can indicate that possible changes are made on the right-part of the control. To make it more generalized, we can normalize the control to $(leftControl < rightControl)$ or just simply $AnyControl$ as mentioned earlier. In a similar way, we can custom normalize the control parts of other statements appropriately to their context. For example, the control part in the *for-loop*, *while-loop* or even in the assignment statement (e.g. $x = (x > 0) : 1 : 0$) can be normalized either in the same way, or in different ways using other TXL rules. One has the option of normalizing the different types of controls to different IDs, or to a single ID. For example, the following TXL rule normalizes the control part of all the *if-then-else* statements to the specific ID *AnyIfControl*.

```
rule ifElseNormalization
     replace $ [statement]
         'if ( Expr1 [expr])
             ThenPart [statement]
        OptElsePart [opt else_statement]
     by
         'if ('AnyIfControl)
             ThenPart
          OptElsePart
end rule
```

With this rule, the code segment of Table 2 (left-column) will be transformed to:

```
if (AnyIfControl)
   m = (y + c)-x;
else
   m = y;
```

Such a normalization of the control part allows arbitrary changes in the control conditions of the copy/pasted segments and our method can efficiently detect them as clones.

As in Basit et al. [4], we also allow for flexible tokenization. We provide an option for equating different token classes, for example to assign the same ID to different but similar data types such as *int, short, long, float, double* depending on user choice.

7. Flexible Code Filtering

Using agile parsing and source transformation we can also efficiently filter out code statements from potential clones according to user preferences. We can filter at any stage, either while extracting potential clones or separately following extraction. While extracting, we can use agile parsing and island grammars to filter out uninteresting statements from potential clones.

To filter out a certain type of statements after extracting potential clones, we can use TXL rules. Filtering out statements is simple and efficient in TXL, simply replacing the uninteresting statement by an empty one. For example, declaration and initialization statements are not major factors in affecting the logic of a code segment and thus could be ignored before comparison using a TXL rule to remove them. In a similar way, when searching for function clones, the function name and parameters can often be ignored. A sample TXL rule for filtering out all the declaration statement could be as follows:

```
rule declarationFiltering
     replace [repeat declaration_and_statement]
         DeclarationPart [declaration]
         Rest [repeat declaration_and_statement]
     by
         Rest
 nd rule
```

8. Comparing the Potential Clones

Once the potential clones are extracted with preprocessing (with or without flexible pretty-printing, code normalization and filtering), they are fed to a comparison algorithm. In this paper, we have used a Longest Common Subsequence (LCS) algorithm for comparing the text lines of potential clones. The details of the algorithm are described elsewhere [17, 16], here we only provide an example of what it does and how it helps in finding clones. The LCS algorithm takes two sequences of items (each item is considered as string) as input and produces the longest sequence of items that is present in both sequences in the same order. For example, consider the following two sequences of items where each item/letter represents a string:

```
Sequence 1: a b c d f g h j q z
Sequence 2: a b c d e f g i j k r x y z
```

The LCS algorithm will produce a new sequence $I < S >$ which can be obtained from the first sequence by deleting some items, and from the second sequence by deleting some others. The condition is that $length(I < S >)$ should be as long as possible. For the above two sequences, we find $I < S >$ as *a b c d f g j z*. Once the sequence is determined, we use the number of unique items for both sequences as a measure of similarity. In the following, we discuss how this algorithm helps us in finding clones. The algorithm is similar to the one employed by the Unix *diff* utility that we have used in our previous work, but has been reimplemented to be more efficient in our application.

To determine whether two potential clones really are clones of each other, we compare their pretty-printed and normalized sequences of text lines as items using LCS. Once we get the longest common subsequence of the two sequences, we determine the number of unique items in

Table 3. Two Function Clones

Item No.	Sequence 1 (Original Segment)	Sequence 2 (Copied and Edited Segment)	Similarity
1	void	void	1
2	sumTimes	sumTimes	1
3	(int n) {	(int n) {	1
4	float sum=	float sum=	1
5	0.0;	0.0;	1
6	double product =	double product =	1
7	1.0;	1.0;	1
8	for (	for (	1
9	int i=1;	int i=1;	1
10	i<=n;	i<=n;	1
11	i++){	i++){	1
12	sum=	sum=	1
13	**sum + i;**	**sum + (i * i);**	**0**
14	product=	product=	1
15	**product * i;**	**product * (i * i);**	**0**
16	fun	fun	1
17	(sum, product);	(sum, product);	1
18	} }	} }	1
	Total Items = 18 Unique Items = 2 UPI = 11.11%	Total Items =18 Unique Items = 2 UPI = 11.11%	

each potential clone. For an item to be declared "common" between the two potential clone sequences, it is not enough for it to occur in both sequences; it has to be an item that occurs in the longest common subsequence of the two sequences. Correspondingly, the items that have to be deleted from the sequences to generate the longest common subsequence are considered to be unique to their respective sequences. In fact, an item that occurs in both sequences might be considered as unique to both of them if it is not part of the longest common subsequence. We then compute the percentage of unique items for each potential clone (i.e., item-sequence) usng the following equation:

$$Unique\,Percentage\,of\,Items\,(UPI) = \frac{No.\,of\,Unique\,Items * 100}{Total\,No.\,of\,Items}$$

If these ratios for both line sequences are either zero or below a certain threshold, the sequences are considered to be clones of each other.

For example, consider the following code segment. Imagine that it is copied and edited in two places (e.g., i is replaced with $i * i$ for the assignment statements in the copied fragment), and assume that the UPI threshold (UPIT) is 30% for considering the two segments as clones.

```
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 }}
```

Following flexible pretty-printing (in this case without code normalization or filtering) of both segments, the corresponding sequences of the fragments are shown in Table 3. After finding the longest common subsequence, we calculate the UPI values for both the sequences. Since both sequences have UPI values (11%) below our assumed threshold (30%), they are considered to be clones.

Table 4. Two Function Non-clones

Item No.	Sequence 1 (Original Segment)	Sequence 2 (Copied and Edited Segment)	Similarity
1	void	void	1
2	**sumTimes**	**sumTimesExtended**	**0**
3	**(int n)** {	**(int n, int m, int x, int y)** {	**0**
4	float sum=	float sum=	1
5	0.0;	0.0;	1
6	double product =	double product =	1
7	1.0;	1.0;	1
		10 Unique Lines	**0**
8/18	for (	for (	1
9/19	int i=1;	int i=1;	1
10/20	i<=n;	i<=n;	1
11/21	i++){	i++){	1
12/22	sum=	sum=	1
13/23	**sum + i;**	**sum + (i * i);**	**0**
14/24	product=	product=	1
15/25	**product * i;**	**product * (i * i);**	**0**
16/26	fun	fun	1
17/27	(sum, product);	(sum, product);	1
18/28	} }	} }	1
	Total Items = 18 Unique Items = 4 UPI = 22.22%	Total Items =28 Unique Items = 14 UPI = 50%	

Now consider another copy with several unique lines added, and some lines modified, including the function name and its parameters. From Table 4, we see that while the UPI value for the original segment (22.22%) is below the assumed UPI threshold (30%), the UPI value for the copied segment (50%) is above the assumed threshold and thus they are not considered to be clones.

From the above examples, we see that the UPI threshold is size-sensitive w.r.t. the number of items in the sequence (i.e., number of lines in the preprocessed potential clones, although one can choose in terms of original lines of code). For example, if UPI threshold is 30%, a potential clone of 10 lines can have a maximum of $((10 * 30/100) = 3)$ three unique lines in compare to its counterpart potential clone. In addition to this size-sensitive threshold, one can also use other size-sensitive thresholds such as *maximum gap size* and *maximum number of gaps* in a sequence.

Clustering and Overall Algorithm: Because the LCS algorithm can only compare two potential clones at a time, in principle each potential clone needs to be compared with all of the others, making the comparisons very expensive. We have used a number of strategies to reduce the number of comparisons, based on the UPI thresholds chosen by the user. If the UPI threshold is 0% (i.e., we are looking for exact matches only), then only potential clones of the exact same size (number of pretty-printed lines) are compared to each other. If UPI threshold is greater than 0%, then a potential clone x is compared to another potential clone y if and only if $size(y)$ in lines is in the range between $size(x) - size(x) * UPIT/100$ and $size(x) + size(x) * UPIT/100$. In essence, this implements dynamic clustering based on the size (i.e., number of lines in the corresponding sequences) of the potential clones and the UPI threshold. In addition, potential clones that are

below a certain threshold size (*Minimum Clone Size*) are either not extracted as potential clones or not compared.

Generating Clone Classes: Once all potential clones are compared and a correspondingly ordered clone pair database is formed, it is a simple step to generate clone classes from the database. If a potential clone P_i, forms a clone pair with another potential clone P_j, then all the other potential clones that form a clone pair with P_j are also included in the clone class and this continues recursively. As we maintain a unique ID for each potential clone and as the clone pair database is ordered by size, one can directly form all clone classes from the database without any computational bottleneck or post-processing. Note that when using this method with a UPI threshold greater than zero, the relation between clones is no longer strictly an equivalence relation since transitivity does not hold, and in the theoretical worst case all instances could be subsumed by the same class. Fortunately, in practice this is not an issue.

Clone Classes Using Exemplars: As in our previous work [10], we also provide the option of applying an even more efficient method to reduce the number of comparisons and find clone classes directly. Although in theory this approach might seem a little arbitrary and clearly can miss some clones, in practice it is very useful and misses very few. The method uses the first clone of each kind as an *exemplar* or distinguished representative of its clone class. When two potential clones x and y are compared and found to be clones, y is marked as being in the class x. That is, x is considered to be the exemplar for the pair. After this finding y is removed from the comparison set and never compared to anything else. Instead, x is compared to all other potential clones of the same cluster and any other matching potential clones are directly added to the class of x. For systems rich in clones, this optimization can reduce comparisons by a large factor. However, once again this is an approximation and in the theoretical worst case it might be possible that x is an exemplar of both y and z even though y and z should not form a clone pair.

Time and Space Complexities: While it is hard to estimate the exact computational and space complexities of this multi-phase detection approach, we can provide an overall estimate. Extraction of potential clones with flexible pretty-printing, code normalization and filtering using the TXL parser and transformation rules is clearly linear in time and space over the total size of the system, requiring three linear passes, one for parsing, one for normalization and filtering, and one for extraction.

Thus the only real performance issue is the comparison of extracted potential clones. In the worst case, one can artificially create a scenario where every potential clone must be compared with every other, requiring quadratic time and linear space in the number of potential clones. In the best case, with the optimizations above, if full code normalization allows us to use direct text comparison of only equally sized potential clones then we can do the whole set of comparisons in linear time. In practice the usual case lies somewhere between, closer to linear than quadratic since the dynamic clusters tend to be very small.

When full flexible code normalization is used, we can use exact text line comparison between potential clone pairs and the time for pair comparison is linear in the size of the potential clones. In the case where differences are allowed, the LCS algorithm used in our prototype for individual pair comparisons has a quadratic worst case time and space complexity. However, in our method the potential clones to be compared are orders of magnitude smaller than the entire system, and the individual comparison time and space is effectively approximated by a small constant per individual comparison. For example, when we had 125 potential clones to compare for *Weltab* using flexible pretty-printing of some statements (c.f., Section 10), there were 15,500 potential comparisons to make. However, using our method with a UPI threshold of 0%, it required only 139 actual comparisons. Further, when we increased the threshold to 10%, it still required only 295 actual comparisons, when the threshold was 20%, 464 actual comparisons, and when the threshold was 30%, 581 actual comparisons. For the small/medium-sized systems we experimented with, no entire computation took more than several seconds.

9. Output Generation

Our framework provides results in two different representations. The user can choose either one of the two or both. The first one is the traditional textual report of the clone class information where each clone class is shown with the corresponding file name and line numbers of the code segments, derived from the source file and line number annotations of the potential clones. The second is the visual representation as of our previous approach [10], which generates an HTML page showing the first code segment as an exemplar for each clone class. Each of the clone classes is linked to a number of secondary clone report pages that shows the other members of the same class. It is also possible to see the similarity values for each pair in the class.

10. Experimental Results

Thus far we have evaluated our approach on two small to medium-size C programs with promising results. The first program was *Abyss* [1], a small web server written in approximately 1,500 lines of C code. The second program was *Weltab* [6] , which is an election results program of approximately 11,000 lines. We have chosen these two systems as our first testbed for two reasons. First, in this first test we want to manually verify the clones found, which is obviously difficult or impossible with large systems. Second, there are already existing published results for *function*

clones available for these systems [31] that we can compare with. Moreover, *Weltab* has already been used in a well known tool comparison experiment [7].

Although our method can be used for finding clones of any granularity (c.f., Section 4), in our first experiment we have considered only clones of function granularity since there are already detailed published results available (with file and function names of all clones found). Tairas and Gray [31] report exact function clones in these systems allowing differences in function names and data types only.

For *Abyss* there are two previously reported clone classes/pairs (Class 1: *ConfGetToken* in *conf.c* and *GetToken* in *http.c*, and Class 2: *ThreadRun* in *thread.c* and *ThreadStop* in *thread.c*). Using our new method we accurately found both of these two classes and one additional valid near-miss clone class (*ConfNextToken* in *conf.c* and *NextToken* in *http.c*) not previously reported.

In case of *Weltab* we have obtained eight exact match clone classes from 27 clone pairs using standard pretty-printing whereas Tairas and Gray [31] obtain only five exact match clone classes (which they report as four groups since they allow function name changes). The additional clone pairs/classes we obtained are (*prtpag* in *ejcn88.c* and *lans.c*), (*whoentrer* in *poll.c* and *spol.c*) and (*shead* in *samp.c* and *sped.c*). We confirmed our additional findings by checking the results submitted to the Bellon et al. tool comparison experiment [6, 7] by a metrics-based tool owner. This is interesting because in this case an AST-based technique did not find even exact clones found by our method (although of course, other AST-based techniques might accurately detect these clones). By relaxing data types, they were able to detect three more classes that we also detected using a UPI threshold of 5%.

With *Weltab* we have also experimented with varying different options to assess their effect in finding clone pairs and clone classes. In Table 5, we provide the the number of clone pairs and clone classes found from *Weltab* using our method depending on the different options we used. The first column of the table shows the UPI threshold we used, the second column shows the clone pairs (CP) and clone classes (CC) found when only standard pretty-printing option is used (std. PP), the third column (Flex. PP) shows *CP* and *CC* when we used flexible pretty-printing for *assignment*, *if* and *for loop* statements, the fourth column (Ctl.Norm) shows *CP* and *CC* when we normalize the control parts of *if*, *for loop* and *while loop* statements, the fifth column (FunDef. Norm) shows the *CP* and *CC* when we normalized/filtered function definition names and the last column (ExpR. Norm) shows *CP* and *CC* when we have normalized the right part of *assignment* statements.

From the table (first row), we see that when we use 0% UPI threshold (i.e., looking for exact match), both standard pretty-printing and flexible pretty-printing return the same number of clone pairs and classes since for exact match clones flexible pretty-printing does not have any effect. However, when we normalize the control part of some statements, we get two more clone pairs and one more clone class. When we normalize the function definition names, we get the same number of clone pairs and one less clone class. This is because, due to the normalizing of function names, two different clone classes become one clone class. On the other hand, when we normalize the right-part of *assignment* statements, we get five more clone pairs and two more clone classes. When we increase the UPI threshold, we get more clone pairs and classes. Given the fact that we are only looking for function clones, not all the potential changes can be captured with code normalization or filtering, so we have tried using different UPI thresholds also with good results.

Table 5. Clone Pairs/Classes from Weltab

UPIT	Std. PP		Flex. PP		Ctl. Norm.		FunDef.Norm		ExpR.Norm	
	CP	CC	CP	CC	CP	CC	CP	CC	CP	CC
0%	27	8	27	8	29	9	27	7	32	10
5%	42	12	44	11	44	12	42	11	42	12
10%	45	11	55	15	47	12	47	11	47	12
20%	59	16	66	19	59	16	63	18	64	18
30%	68	20	79	25	70	21	72	22	70	21

In these first tests we have not found any false positives using our method except when we use a very large UPI threshold. Although we have manually examined all the functions of the both systems to assess whether our method has missed any, we of course cannot be completely sure. To further validate the method, we decided to inject a substantial number of new clones of different types using editing scenarios [29] into the systems and attempted to find them using our method. Our method was able to effectively find all injected clones of the different types.

While these initial tests of our method are by no means conclusive, the results are certainly promising, and we look forward to a more extensive controlled experiment comparing our method with others.

11. Related Work

Most text-based approaches [13, 33] are related to our work in the sense that like them we also find clones by comparing program text. Although many of them can find near-miss clones, these approaches do not find syntactic clones and do not provide for accurate approximation in any way similar to our flexible pretty-printing, normalization and filtering.

Most lexical approaches [20, 4, 3, 25] (also called token-based approaches) are related to our work in the sense that like them we also use a similar sequence matching algorithm, and like them, we can also apply token transformation on the input. However, we do not use a generalized

tokenization of identifiers and literals (e.g., normalizing all identifiers to a unique id) as they do. Moreover, special treatment or post-processing is required to find syntactic clones or gapped clones with such tools, which calls for the use of further "helper" tools such as *CLICS* [22] or extended implementation in *Gemini* [32]. We deal with these issues easily in our pre-processing and comparison phases.

Syntactical approaches [5, 18, 24, 31, 14] (also called tree-based approaches) are also related to ours in the sense that they are also parser-based and are aimed at syntactic clones. However, these are heavily dependent on fully-fledged parsers and find clones on ASTs or sequences of AST nodes (suffix trees) [24, 31], whereas we work on (pretty-printed) program text. Our method adds more flexible and restricted code normalization and filtering, and provides for the post-normalization inexact matching necessary to find many non-structural near-misses. *Asta* [14] can find near-miss clones based on structural abstraction much like ours, but using arbitrary pattern matching on ASTs.

For a detailed introduction to all of the various available methods, the reader is referred to our technical report [28] or summary paper [29].

12. Conclusion

Clone detection is an active research area and the literature is rich with work on detecting, removing and analyzing clones. In this paper we have presented a new clone detection method based on a two stage approach: identification and normalization of potential clones using flexible pretty-printing and code normalization, followed by simple text-line comparison of potential clones using dynamic clustering. Early experiments demonstrate that this new method can do at least as well as existing methods in finding and classifying function clones in C. In addition to continuing our empirical validation for larger and more challenging C systems, in future we will be exploring the application of our method to different languages, and designing a new mutation-based [2] controlled experiment to compare it with other methods using synthetically generated clones of different types and granularities.

Acknowledgements: The authors would like to thank Robert Tairas and the four anonymous reviewers for their valuable comments, suggestions and corrections in improving the paper. Thanks also to Nikita Synytskyy and Thomas R. Dean for providing us with resources from their earlier work on clones. This work is supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] The Abyss: http:// abyss.sourceforge.net/ (December, 2007)
[2] J. H. Andrews, L. C. Briand and Y. Labiche. Is Mutation an Appropriate Tool for Testing Experiments? In *ICSE*, pp. 402-411, 2005.
[3] B. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *WCRE*, pp. 86-95, 1995.
[4] H. Basit, S. Pugliesi, W. Smyth, A. Turpin and S.Jarzabek. Efficient Token Based Clone Detection with Flexible Tokenization. In *ESEC/FSE*, pp. 513-515, 2007.
[5] I. Baxter, A. Yahin, L. Moura and M. Anna. Clone Detection Using Abstract Syntax Trees. In *ICSM*, pp. 368-377, 1998.
[6] S. Bellon and R. Koschke. Detection of Software Clones: Tool Comparison Experiment. URL: http://www.bauhaus-stuttgart.de/clones/ (December, 2007).
[7] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE TSE*, 33(9):577-591, 2007.
[8] The TIL Chairmarks. *www.program-transformation.org/Sts/TILChairmarks*, 2008.
[9] J.R. Cordy. The TXL source transformation language. In *Science of Computer Programming*, 61(3):190-210, 2006.
[10] J.R. Cordy, T.R. Dean and N. Synytskyy. Practical Language-Independent Detection of Near-Miss Clones. In *CASCON*, pp. 29-40, 2004.
[11] T.R. Dean, J.R. Cordy, A.J. Malton and K.A. Schneider. Agile Parsing in TXL. *J. ASE*, 10(4):311-336, 2003.
[12] A. van Deursen and T. Kuipers. Building Documentation Generators, In *ICSM*, pp. 40-49, 1999.
[13] S. Ducasse, M. Rieger and S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *ICSM*, pp. 109-118, 1999.
[14] W. Evans and C. Fraser. Clone Detection via Structural Abstraction. In *WCRE*, pp. 150-159, 2007.
[15] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.
[16] J. W. Hunt and M. D. McIlroy. *An Algorithm for Differential File Comparison*. Technical Report 41, Bell Laboratories, 1976.
[17] J. W. Hunt and T. G. Szymanski. A Fast Algorithm for Computing Longest Common Subsequences. *Comm. ACM*, 20(5):350-353, 1977.
[18] L. Jiang, G. Misherghi, Z. Su and S. Glondu. DECKARD: Scalable and Accurate Tree-based Detection of Code Clones. In *ICSE*, pp. 96-105, 2007.
[19] J. Johnson. Visualizing Textual Redundancy in Legacy Source. In *CASCON*, pp. 171-183, 1994.
[20] T. Kamiya, S. Kusumoto and K. Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE TSE*, 28(7):654-670, 2002.
[21] C. Kapser and M. Godfrey. "Cloning Considered Harmful" Considered Harmful. In *WCRE*, pp. 19-28, 2006.
[22] C. Kapser and M. Godfrey. Supporting the Analysis of Clones in Software Systems: A Case Study. *JSME: Research and Practice*, 18(2):61-82, 2006.
[23] M. Kim and G. Murphy. An Empirical Study of Code Clone Genealogies. In *FSE*, pp. 187-196, 2005.
[24] R. Koschke, R. Falke and P. Frenzel. Clone Detection Using Abstract Syntax Suffix Trees. In *WCRE*, pp. 253-262, 2006.
[25] Z. Li, S. Lu, S. Myagmar and Y. Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE TSE*, 32(3):176-192, 2006.
[26] J. Mayrand, C. Leblanc and E. Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *ICSM*, pp. 244-253, 1996.
[27] L. Moonen. Generating Robust Parsers using Island Grammars. In *WCRE*, pp. 13-22, 2001.
[28] C.K. Roy and J.R. Cordy. *A Survey on Software Clone Detection Research*. School of Computing TR 2007-541, Queen's University, 115 pp., 2007.
[29] C.K. Roy and J.R. Cordy. Scenario-Based Comparison of Clone Detection Techniques. In *ICPC*, 10 pp., 2008.
[30] F.V. Rysselberghe and S. Demeyer. Evaluating Clone Detection Techniques. In *ELISA*, 12 pp., 2003.
[31] R. Tairas and J. Gray. Phoenix-Based Clone Detection Using Suffix Trees. In *ACM-SE*, pp. 679-684, 2006.
[32] Y. Ueda, T. Kamiya, S. Kusumoto and K. Inoue. On Detection of Gapped Code Clones Using Gap Locations. In *APSEC*, pp. 327-336, 2002.
[33] R. Wettel and R. Marinescu. Archeology of Code Duplication: Recovering Duplication Chains From Small Duplication Fragments. In *SYNASC*, 8 pp., 2005.