KClone: A Proposed Approach to Fast Precise Code Clone Detection

Yue Jia[1], David Binkley[2], Mark Harman[1], Jens Krinke[1] and Makoto Matsushita [3]

[1]King's College London
Strand, London
WC2R 2LS, UK

[2]Loyola College in Maryland
4501 North Charles Street
Baltimore, MD 21210-2699, USA

[3] Osaka University
1-5 Yamadaoka, Suita
Osaka 565-0871, Japan

Abstract

In all applications of clone detection it is important to have precise and efficient clone identification algorithms. This paper proposes and outlines a new algorithm, KClone for clone detection that incorporates a novel combination of lexical and local dependence analysis to achieve precision, while retaining speed. The paper also reports on the initial results of a case study using an implementation of KClone with which we have been experimenting. The results indicate the ability of KClone to find types-1,2, and 3 clones compared to token-based and PDG-based techniques. The paper also reports results of an initial empirical study of the performance of KClone compared to CCFinderX.

1 Introduction

Previous work has concentrated on three types of clones, known as Type-1, Type-2 and Type-3 [2]. Type-1 clones are those created by verbatim copy-and-paste. Type-2 clones augment this with some vairable, type or function identifiers changes but no statement additions, deletions, or reordering. Type-3 clones allow additions, deletions, or reorderings.

Previous work can be categorised according to how it handles these three types of clone:

- **Type-1:** This kind of clone can be detected easily.
- **Type-2:** This kind of clone can also be detected but not as easily. A widely studied subset of this type involves two code fragments f_1 and f_2 where uniformly replacing identifiers in f_1 yields f_2.
- **Type-3:** One or more statements can be modified, added, or removed. Furthermore, the structure of code fragment may be changed and it may even look or behave slightly different from the original. This kind of clone is hard to detect.

Increasing degrees of algorithmic sophistication are required to identify the higher type clones. This increased algorithmic sophistication has, hitherto, brought with it a computational cost.

Definition 1 (Inclulsive) *A clone detection approach, C, is inclusive with respect to a set of n clone detection approaches $C_1, ..., C_n$ if the clones found by C are a superset of those found by $C_1, ..., C_n$.*

Fast algorithms typically fail to identify some Type-2 and most Type-3 clones, but scale to large systems, while those that target Type-3 clones using dependence-based algorithms may find Type-3 clones, but at a high computational cost. Thus, the current state of the art presents the software engineer with a classic 'speed-quality' trade off.

This paper argues that it is possible to have the best of both worlds; fast, inclusive clone detection is achievable, with the result that it will be possible to find Type-1,2 and 3 clones in large real world systems. The paper proposes and outlines a new algorithm, KClone, for clone detection that incorporates a novel combination of lexical and local dependence analysis. The local nature of the dependence analysis allows KClone to achieve inclusiveness without sacrificing speed. In this paper we argue that KClone is inclusive with respect to CCFinderX and Duplix.

Unlike Types-1 and 2, which have a precise definition, Type-3 clones are less precisely defined [2]. For example, there needs to be some limit on the extent to which two fragments of code can differ while still being considered clones of one another. This can be captured using a *statement gap distance* or based on a measure such as the Levenshtein edit distance[6].

For example, Figure 1 shows an example clone class that KClone detected in the Java project NetBeans-Javadoc. The three code fragments are selected from three different source files and the duplicated code is indicated by the mark "++". The duplicated code is not exactly the same; some

Code fragment A

```
++ public ServiceType getExecutor(){
       JavadocType.Handle javadocType =
         (JavadocType.Handle)getProperty(PROP_ EXECUTOR);
++     JavadocType type = null;
++     if (javadocType != null){
++       type = (JavadocType)javadocType.getServiceType();}
++     if (type == null){
         if (isWriteExternal()){
           return null;}
++       return(JavadocType)Lookup.getDefault().lookup(org.netbeans.
++         modules.javadoc.settings.ExternalJavadocSettingsService.class);}
++     return type;}
```

Code fragment B

```
++ public ServiceType getExternalExecutorEngine(){
++     ExternalJavadocExecutor service = null;
++     if (executor != null){
++       service = (ExternalJavadocExecutor)executor.getServiceType();}
++     if (service == null){
++      return (ServiceType)Lookup.getDefault().lookup(org.netbeans.
++        modules.javadoc.ExternalJavadocExecutor.class);}
++     return service;}
```

Code fragment C

```
++ public ServiceType getSearchEngine(){
       JavadocSearchType.Handle searchType =
         (JavadocSearchType.Handle)getProperty(PROP_ SEARCH);
++     JavadocSearchType type = null;
++     if (searchType != null){
++       type = (JavadocSearchType)searchType.getServiceType();}
++     if (type == null){
         if (isWriteExternal()){
           return null;}
++       return(JavadocSearchType)Lookup.getDefault().lookup(org.netbeans.
++         modules.javadoc.search.Jdk12SearchType.class);}
++     return type;}
```

Figure 1. An example detected by KClone

variable names and types have been changed, some lines are deleted, and some new lines insert. As a result, these types of clone are difficult to detect.

A good clone detector should scale to large programs, while considering sufficient semantic-level information to detect all three types of clone. This requires that the management of necessary semantic information should be inexpensive in terms of time and memory. KClone aims to achieve this goal.

The rest of this paper is organized as follows. Section 2 describes the general structure of clone detection algorithms and presents an overview of the KClone algorithm proposed in this paper. Section 3 explains the experimental setting as well as the initial results and the paper concludes with Section 4.

2 Algorithm Description

Although various clone detection techniques use different data representations and matching algorithms, they all follow a similar three-step process:

Step 1 **Transform** the code into an internal representation.

Step 2 **Detect** parts that denote clone pairs.

Step 3 **Aggregate** clone pairs into clone classes.

Of these steps, Step 2 is the most time consuming, while the representation has the greatest impact on inclusiveness. The key to KClone's fast and inclusive clone detection is to divide Step 2 into two phases: first, a fast lexical analysis detects *basic clone pairs* (BCPs). Only Type-1 or Type-2 clones are identified by this step. However, the detected contiguous code sequences help detect Type-3 clones. The second phase of the detection extends the BCPs into (final) clone pairs while avoiding the construction of 'expensive' data structures such as Abstract Syntax Trees (ASTs) or Program Dependence Graphs (PDGs). Using these two phases, KClone attains the efficiency of token and text-based methods, while attaining the inclusiveness of more semantics-based methods.

KClone's three steps are now described in more detail. First, Step 1 filters out *uninteresting* statements and then extracts necessary "light-weight" information from the remaining statements. Uninteresting statements include comments, blank lines, C/C++ preprocessor directives (*e.g.*, #include) and in Java the directives "import" and "package". In this paper, the term *statement* is used to refer to a single textual line of code, rather than a programming language statement construct in the formal language sense of the word.

From the remaining *interesting* statements, three values are extracted: *structure-code, control links*, and *data dependence links*. Structure-code represents the structure of each statement by storing the token identifiers of each lexeme found in a statement. This effectively captures the code's structure, while allowing for certain common changes such as variable renamings. For each statement, the control-link identifies the previous and next statements (in the source code ignoring nesting, etc.) that contain a control keyword (*e.g.*, while).

Finally, for each token that represents a variable, the data-dependence-link (hereafter simply data-link) identifies the previous and next statement that references (uses or defines) the same variable. This is a local computation of dependence, from which KClone derives comparable inclusiveness to dependence-graph-based approaches to clone detection.

The efficiency of the Step 2 comes from dividing it into two phases: The first phase applies a modified string suffix algorithm and identifies BCPs (basic clone pairs). These continuous statement sequences are sufficient to capture many Type-1 and Type-2 clones. The second phase then extends BCPs to detect Type-3 clones by *growing* each BCP to include 'nearby' statements. This is particularly effective

when the copied code has been edited or has had intervening statements interjected.

In more detail, Phase 1 of Step 2 identifies BCPs created by simple copy-and-paste operations. These are often the core of the clones where the copied code is subsequently edited. Phase 1 begins with an adaptation of the suffix comparison algorithm used widely in text searching [3, 5, 7] as well as text and token based clone detectors [1, 4]. The adaptation, uses token identifiers in place of the tokens as the comparison unit. It also uses a fixed length suffix for each statement. In the current implementation, the suffix is composed of the token values from five contiguous statements. When two statements have the same suffix then they form a BCP.

Phase 2 extends the results of Phase 1 to detect larger Type-1 and 2 clones as well as Type-3 clones. This is done using the control and data links to efficiently bypass potentially interjected or modified code. Phase 2 is invoked for each BCP and repeatedly extends the fragments that make up the BCP using one of six extension functions, including *extend-backwards*, *extend-forwards*, *extend-control-back*, *extend-control-forward*, *extend-data-back*, and *extend-data-forward* until no functions can be applied to BCPs. Extension terminates in one of three situations: the (structure of the) linked-to statements are not equal, the linked-to statements are more than a specified distance away from the current clone, or when the 'window of text' under consideration reaches the start or end of its file.

For example, the extension function *extend-backwards* checks the statements immediately before the two code fragments A and B of a BCP. If they match (*i.e.*, they have the same *structure-code*), then the algorithm adds these statements to the clone pair.

The four 'more complex' extension functions require a distance function, *dist* that returns the difference in the line numbers of two statements (or maxint when one of the statements is NULL). This occurs only if one of the next lines occurs at the end of the file or previous lines at the beginning of the file.

These four functions are also parameterized by one of more of three integers: cg, dg, and w. The control-gap, cg, is a threshold controlling the largest gap that the extension will consider when following a control links. Similarly, the data-gap, dg, is the largest gap considered when following a data link. Finally, window size, w, controls the number of statements considered when performing the data extension. This final parameter limits the extension as follows: when extending a clone pair backwards (forwards), only data-links of variables in the first (last) w statements are considered. In the experiment, the values $cg = 3$, $dg = 5$, and $w = 5$ were used.

The four remaining functions for extending a clone pair are all similar. *Control-back* for extending a clone pair backwards using the control link is considered as an example. It considers control statements within the control-gap distance cg of the first statements of two code fragments A and B. Looking back using the control-link, if a match is found the clone pair is extended back to include the matching statements. If multiple matches exist, the 'farthest' (from the beginning of the clone pair) is chosen.

Conceptually, the efficiency of KClone can be attributed to a combination of lexical and dependence techniques. To begin with, the use of 'light weight' lexical analysis supports fast detection of Type-1 and Type-2 clones (similar to other fast lexical clone detectors). For example, KClone first uses fixed-length suffix analysis to find (small) BCPs, which are then extended to final clone pairs. Previous applications of suffix analysis have not bounded the suffix length. This can be very expensive when a large part of the program is a potential suffix.

The lexical analysis also gathers sufficient dependence information in the control and data links to quickly detect Type-3 clones, giving precision at reasonable cost. Although the search for Type-3 clones inherently involves $O(n^2)$ comparisons to correctly jump over interjected code, KClone exploits the control and data links to reduce the value of n. For example, in the case of control predicates n is not 'all statements within the gap distance', but only 'all control statements within the gap distance'. This allows significant code to be ignored during extension.

Example. Figure 2 shows an example Type-3 clone found by KClone in the 11KLOC C program WelTab. Lines marked by ++ denote cloned code. In this case, after copy and paste, Line 10 of code Fragment A was edited. Depending on the gap distance used, textual-based approaches may miss Lines 12 and 13.

When KClone is applied to this example, it first identifies Lines 1 to 5 of both code fragments as (structurally) equivalent and thus as a BCP. This BCP is expanded to cover Lines 1 to 9 by the function extend-forward which stops because the two Lines numbered 10 in the two fragments are not equal. Next, the function extend-control-forward identifies, through the control-link, that Line 11 of Fragment A matches Line 12 of Fragment B. After extending the clone pair to include these two statements, extend-forward adds Line 12 of A and Line 13 of B. At this point, the expansion stops because none of the surrounding code (not shown in figure 2 for brevity) is structurally equivalent. Note that in this example, even if the control-links are ignored, the extension would still have uncovered that Lines 11-12 of A match Lines 12-13 of B through the data-link for variable buffer.

After dependence analysis the third and final step aggregates clone pairs into clone classes. This can be done in several ways. KClone uses transitive paring to group clones together: if code Fragments A and B form a clone pair and

Code Fragment A	Code Fragment B

1 ++ if (buffer[0] != '1'
2 ++ && buffer[0] != ' '
3 ++ && buffer[0] != '0'
4 ++ && buffer[0] != '+') {
5 ++ if (nread != 1) printf("\n");
6 ++ printf("%s",buffer);
7 ++ nwrite++;
8 ++ };
9 ++ if (buffer[0] == '1')
10 printf("\f ");
11 ++ if (buffer[0] == " ') {
12 ++ if (nread != 1) printf("\n");
13
	1 ++ if (buffer[0] != '1'
	2 ++ && buffer[0] != ' '
	3 ++ && buffer[0] != '0'
	4 ++ && buffer[0] != '+') {
	5 ++ if (nread != 1) printf("\n");
	6 ++ printf("%s",buffer);
	7 ++ nwrite++;
	8 ++ };
	9 ++ if (buffer[0] == '1'){
	10 if (nread != 1) printf("\f ");
	11 };
	12 ++ if (buffer[0] == " ') {
	13 ++ if (nread != 1) printf("\n");

Figure 2. Software clone example illustrating the dependence analysis

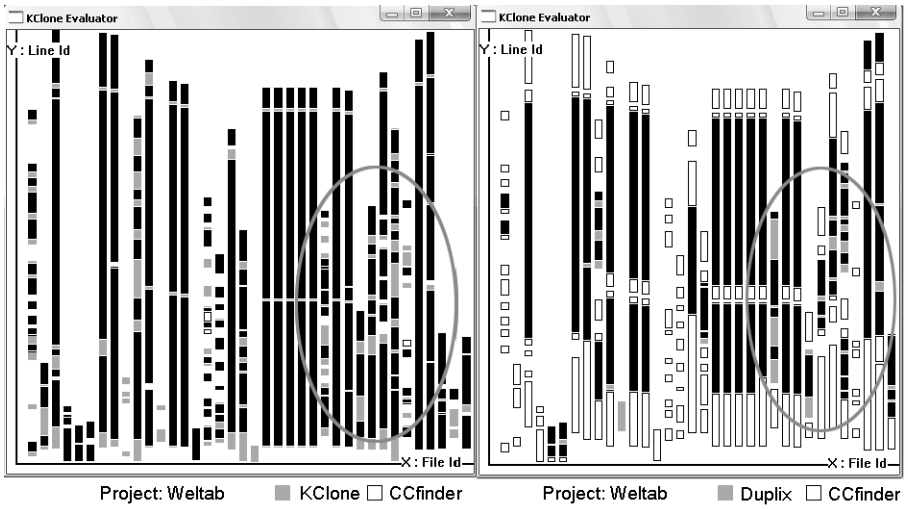

Figure 3. Example of un-proposed clone. (Black lines are common clone lines, gray lines are found only by the first detector, and those in outline are found only by the second.)

code Fragments B and C form a clone pair, then A, B, and C are placed in the same clone class. Forming clone classes benefits programmers, by grouping common clones.

3 Illustration of Results

KClone is implemented in C and finds clones in C, C++, and Java programs. We plan to release a copy of KClone to the community in due course. After loading a project's source into memory, KClone preprocesses all the files before running first the syntactic and then the dependence analysis. Detection outputs all clone pairs and all the clone classes. This section reports initial result concerning KClone's performance with regard to inclusion, speed and memory.

3.1 Inclusion Study

To see the way in which KClone achieves inclusiveness, we performed a comparison between CCFinder and Duplix using the WelTab program and the benchmark clone pairs reported by Bellon and Koschke [2] (see Figure 3). Two observations can be made. First, the grey bars in the two show clone lines found by either KClone (on the left) or by Duplix (on the right), but missed by CCFinder. As expected, given the past performance of Duplix, manual inspection of these clone lines confirmed that they all correspond to Type-3 clones. The results provide evidence that KClone is good at identifying these hard-to-identify Type-3 clones.

The second observation is that there are very few clone lines shown in outline on the left of Figure 3, but a considerable number on the right. To understand the cause of this

Table 1. Running time and memory-consuming of KClone. (Interesting LoC excludes comments, blank lines, and directives (*e.g.*, #define and import)

Program	LoC	I LoC	Peak Memory (MB)		Running time (secs)	
			CCFinderX	KClone	CCFinderX	KClone
WelTab	11K	9K	243	7	5	4
cook	80K	45K	244	30	18	10
snns	115K	70K	243	48	30	15
netbeans-javadoc	19K	8K	244	9	5	2
eclipse-ant	35K	15K	245	13	10	6
eclipse-jdcore	148K	94K	243	67	55	20

difference, it is first important to note that, while Duplix is good at finding Type-3 clones, it is weak at finding Type-2 clones. The clones lines shown in outline on the right of Figure 3 are largely Type-2 clones. However, the absence of similar regions shown in outline on the left of Figure 3 indicates that these clone lines are detected by KClone; thus, Figure 3 illustrates that KClone is good at finding all types of clone.

3.2 Speed and Memory Study

Bellon and Koschke report the worst case for each tool they study [2]. Paraphrasing their results, the metric-based techniques are the most efficient: for even the largest of programs considered they take no more than 5 seconds and consume no more than 50MB of memory. The text-based and token-based techniques are also very efficient. They take 10 to 50 seconds to analyze around 100K LOC and consume a similar 50MB of memory. However the AST and PDG based methods are comparatively slow and require significant memory. For example, when applied to the 115KLoC program SNNS, CLoneDR (an AST based approach) takes 3 hours and uses 628MB of memory, while Duplix (a PDG based approach) takes 63 hours and uses 64 MB of memory. This reflects their need to build 'heavy weight' data structures.

To provide a concrete evaluation of the performance of KClone, the memory and run-time for CCFinderX and KClone were gathered. Beyond the pragmatic reason that it is the only publicly available clone detector studied, CCFinderX is also a good choice because of its good performance results. Only the metric-based approaches are faster, but they tend to uncover considerably fewer clones.

The comparison ran KClone and CCFinderX on the same hardware and in the same software environment. It measured the memory and time required by each. Table 1 presents the results. The memory demand of KClone is clearly a function of the input size and considerably less than that of CCFinderX. From the run-times presented in Table 1 KClone takes, on average, about half of the time of CCFinderX. This is largely due to the combination of quickly finding basic clone pairs using the modified suffix algorithm and then expanding them using the control and data links. Given that its inclusiveness rivals considerably more expensive clone detectors, we believe that these results provide evidence to support the claim that KClone represents an attractive combination of inclusion and efficiency.

4 Conclusion

This paper proposes and outlines an algorithm capable of detecting all three clone types studied in the literature. The novel aspect of the algorithm is the exploitation of both textual and dependence information. The key benefit of this combination is an improvement in both inclusiveness of all clone types and performance. For example, the algorithm can quickly detect Type-3 clones which are normally only detected by slow, semantic, clone detection techniques.

References

[1] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Second Working Conference on Reverse Engineering*, pages 86–95, Los Alamitos, California, 1995.

[2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[3] M. Crochemore and W. Rytter. *Jewels of stringology*. World Scientific Publishing, River Edge, NJ, 2003.

[4] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[5] P. Koa and S. Alurua. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3:143–156, 2004.

[6] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet Physics-Doklandy*, volume 10, 1966.

[7] P. Weiner. Linear pattern matching algorithms. In *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)*, 1973.