Detection and Analysis of Near-Miss Software Clones

Chanchal K. Roy
School of Computing, Queen's University, Kingston, ON, Canada K7L 3N6
croy@cs.queensu.ca

Abstract

Software clones are considered harmful in software maintenance and evolution. However, despite a decade of active research, there is a marked lack of work in the detection and analysis of near-miss software clones, those where minor to extensive modifications have been made to the copied fragments. In this thesis, we advance the state-of-the-art in clone detection and analysis in several ways. First, we develop a hybrid clone detection method. Second, we address the decade of vagueness in clone definition by proposing a metamodel of clone types. Third, we conduct a scenario-based comparison and evaluation of all of the currently available clone detection techniques and tools. Fourth, in order to evaluate and compare the available tools in a realistic setting, we develop a mutation-based framework that automatically and efficiently measures (and compares) the recall and precision of clone detection tools. Fifth, we conduct a large scale empirical study of cloning in open source systems.

1. Introduction

Reusing code fragments by copying and pasting with or without minor adaptation is a common activity in software development. As a result software systems often contain sections of code that are very similar, called *software clones*. Previous research shows that a significant fraction of the code in a typical software system has been cloned [11] and that such clones are considered harmful in software maintenance and evolution [4]. For example, if a bug is detected in a code fragment, all fragments similar to it should be checked for the same bug.

Many other software engineering tasks, such as program understanding (clones may carry domain knowledge), code quality analysis (fewer clones may mean better quality code), aspect mining (clones may indicate the presence of an aspect), plagiarism detection, copyright infringement investigation, software evolution analysis, code compaction (for example, in mobile devices), virus detection, and bug detection may require the extraction of syntactically or semantically similar code fragments, making clone detection an important and valuable part of software analysis [11].

However, despite a decade of active research, there has been a marked lack of work in the detection and analysis of software clones, especially with respect to near-miss software clones. In this thesis, we advance the state-of-the-art in clone detection research in the context of both exact and near-miss software clones.

The rest of the paper is organized as follows. Following a short introduction to general types of clones in Section 2, we provide our hybrid clone detection method in Section 3. In Section 4 we propose a metamodel of clone types that is used as the foundation of the other parts of this thesis. While Section 5 presents a scenario-based qualitative comparison and evaluation of all the existing clone detection techniques and tools, Section 6 provides a mutation-based framework that automatically and efficiently evaluates and compares clone detection tools in a realistic setting. Section 7 presents and analyzes the findings of an empirical study that examines the cloning status of several large systems in several different languages. Finally, Section 8 concludes the paper with directions for future research.

For each of the parts, we first describe the technical problem along with its importance and relevance. We then state the research question or hypothesis along with the proposed solution and methods. Finally, we present the results we have obtained so far with corresponding citations.

2. Background

Definition 1: Code Fragment. A code fragment (CF) is any sequence of code lines (with or without comments). It can be of any granularity, e.g., function definition, begin-end block, or sequence of statements.

Definition 2: Software 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) = f(CF2)* where *f* is the similarity function (see *clone types* below). Two fragments that are similar to each other form a *clone pair* $(CF1, CF2)$, and when many fragments are similar, they form a *clone class*. In the following we provide the general types of clones from the literature [11].

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

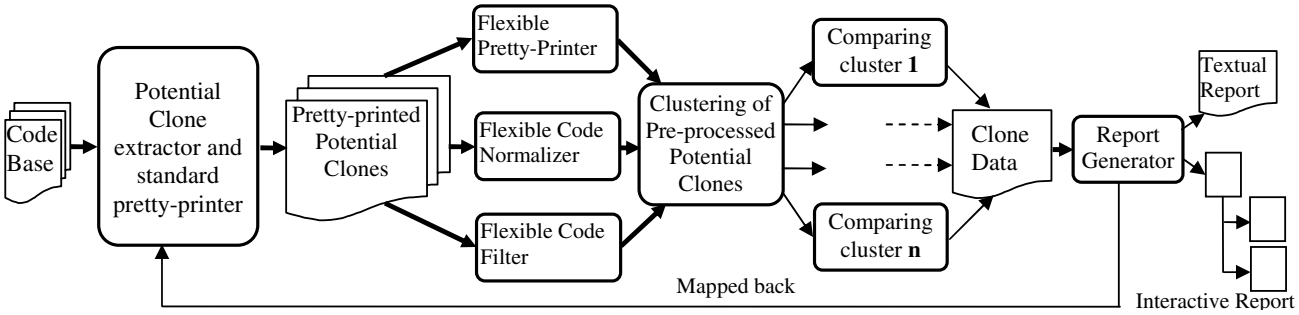

Figure 1. NICAD's Clone Detection Process

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. The NICAD Tool

Considering the importance, over the last decade several (semi-)automated techniques for detecting code clones have been proposed [7, 10]. 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 [2]. 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. Experimental results also show that parser-based techniques give low recall [2]. Moreover, neither text-based nor parser-based techniques have been found to be effective in detecting near-miss clones [2].

In this thesis, we propose a multi-pass hybrid approach, called NICAD [9, 12] which is parser-based and language-specific but reasonably lightweight and detects both exact and near-miss clones with high precision and recall, and with reasonable performance. Figure 1 represents a conceptual diagram of our new clone detection process. NICAD works in three phases:

Standard Pretty-Printing and Extraction: This is the first phase where all potential clones (code fragments of the target kind) are identified, pretty-printed and extracted from the subject code base. During the extraction, they are also automatically stripped of formatting and comments and pretty-printed by TXL [3] according to the language grammar's formatting cues. Standard pretty-printing ensures consistent layout and spacing of code for later comparisons.

Extracted potential clones can be further processed with three more flexible options, (1) *Flexible pretty-printing*-in which language and (possibly) project specific special pretty-printing is applied on the potential clones. This 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, (2) *Flexible code normalization*-in which we can easily normalize parts of a statement (or whole statements of a given type) to ignore editing differences using TXL's transformation rules, and (3) *Flexible code filtering*-in which we can efficiently filter out unimportant code statements from potential clones according to user preferences.

Clustering and Comparison: NICAD compares the pretty-printed potential clones line-by-line textwise using a longest common subsequence (LCS) algorithm similar to the Unix *diff* utility. To determine whether two potential clones really are clones of each other, we compare their pretty-printed (and optionally normalized/filtered) text lines and use the number of unique lines in each as a measure of similarity/dissimilarity. In particular, we compute the size-sensitive Unique Percentage of Items (UPI) for each potential clone using the equation (for details see [9]):

$$UPI = \frac{No.\ of\ Unique\ Items * 100}{Total\ No.\ of\ Items}$$

If the UPI for both line sequences is zero or below a certain predefined UPI threshold (UPIT), the potential clones are considered to be clones. The number of comparisons is optimized using an exemplar-based technique that builds clone classes directly by choosing the largest unclassified potential clone as an exemplar and comparing similarly-sized potential clones to it depending on the UPI.

Reporting: Results from the *Comparison* phase are reported in both XML database form with detailed information about each clone class and as an interactive HTML website where original code fragments are shown in HTML pages to assist hand validation.

NICAD is evaluated in three phases: first, with an early experiment [9], second, with a mutation-based evaluation framework [5] and third, with large scale open source systems of different languages [8, 12].

4. Metamodel of Clone Types

The definition of software clone in the literature [5, 11] is inherently vague and in most cases are detection-dependent and/or task-specific. To overcome this limitation, we have proposed a metamodel of clone types [5, 10] that models developers' editing activities.

Intuitively, in most cases the "clones" we are looking for are those created as a result of copy/paste/modify actions by programmers. In our work we begin with this assumption, and use it as the basis of a top-down theory of clones, which we have formalized into a taxonomy of the editing actions that a programmer may undertake in the intentional creation of *Type 1* to *Type 4* clones. Our metamodel is not simply guesswork - it is derived from the large body of published work on existing clone definitions, clone types, clone taxonomies and studies of developer copy/paste activities [11]. We have validated the metamodel by studying the copy/paste patterns of function clones [6] found in an empirical study (Section 7). NICAD is then designed to address the different types of clones in the metamodel.

5. Qualitative Comparison and Evaluation

Clone detection techniques are often inadequately evaluated. The general lack of evaluation is exacerbated by the fact that there are no agreed-upon evaluation criteria or representative benchmarks [1]. In particular, accuracy measures such as precision and recall have only been roughly estimated, due both to problems in creating a validated clone benchmark against which tools can be compared, and to the enormous manual effort required to hand check large numbers of candidate clones. In an attempt to compare all clone detection techniques more uniformly, independent of tool availability, implementation limitations or language, we have conducted a scenario-based qualitative approach and organize the large number of techniques and tools in a coherent conceptual framework [7, 10].

First, we perform a classification and overall comparison with respect to a number of facets, each of which has a set of (possibly overlapping) attributes. Second, based on the metamodel introduced in Section 4, we design editing scenarios to create *Type 1*, *Type 2*, *Type 3* and *Type 4* clones. Using these scenarios we qualitatively evaluate the techniques and tools we have previously classified. In particular, we estimate how well the various clone detection techniques may perform based on their *published* properties (either in the corresponding published papers or online documentation). Using this framework, one can efficiently and instantly choose the right set of tools for her purpose or can develop a hybrid clone detection method. The hybrid architecture of NICAD is in fact derived from this framework. Consequently, NICAD was placed one of the best tools available in the framework.

Table 1. Recall and Precision of NICAD (%)

Type	Standard PP		Flexible PP		Full NICAD	
	Rec.	Prec.	Rec.	Prec.	Rec.	Prec.
Type 1	100	100	100	100	100	100
Type 2	29	94	27	94	100	97
Type 3	95	85	94	81	100	96
Type 4	67	81	67	79	77	89
Total Overall	87	90	84	89	96	95

6. Mutation-based Evaluation Framework

Again, the conceptual framework proposed in Section 5 above can only provide qualitative comparison and evaluation of the clone detection techniques and tools rather than quantitative measures. In order to evaluate and compare the available tools in a realistic setting, in this thesis, we develop a mutation / injection-based framework [5] that automatically and efficiently measures (and compares) the recall and precision of clone detection tools for different fine-grained clone types of the proposed metamodel (Section 4).

For each of the fine-grained clone types of the metamodel (Section 4), we design code mutation operators to model each type by mimicking developers' typical editing activities in clone creation using TXL. By using these operators, randomly mutated clone fragments are generated from the original subject code base. These mutated fragments are then injected to the original code based to get thousands of mutated code bases. By tracking the large number of injected artificial clones in these mutated code bases, we then automatically measure how well (i.e., values precision and recall) these known clones are detected by a particular tool (for individual tool evaluation) or group of tools (for comparing different tools).

First, we have evaluated NICAD using this framework (where open source *Weltab* has been used as code base) and found that it gives high precision and recall for different fine-grained types of clones. We have then compared NICAD with two of its variants, the standard and flexible pretty-printing (PP) options with dissimilarity thresholds (Section 3). While the detailed results for each of the fine-grained clone types are available elsewhere [5], in Table 1 we provide the recall and precision of NICAD and its variants for the four general clone types. From the table, we see that full NICAD gives both high precision and recall for *Type 1*, *Type 2* and *Type 3* clones and better than its variants.

Although the framework is initially targeted at evaluating NICAD and its variants, the objective was to design a generic framework for evaluating and comparing clone detection tools by overcoming the known challenges to objective tool comparison experiments. The resulting framework is flexible and adaptable enough to evaluate third party clone detection tools provided that the tools can be run from the command line and that they provide textual report of detected clones with full file name, and begin and end line numbers of the code fragments of found clone pairs.

7. A Large Scale Empirical Study

Again, despite a decade of active research there has been a marked lack of in-depth comparative studies of cloning, particularly in a variety of systems. There have been many empirical studies on cloning, for example every new technique comes with some sort of empirical validation [7], and empirical studies are used when comparing tools [2].

However, in both cases the focus is on validating or comparing the techniques rather than the clone properties of the subject systems themselves. Particular subject systems have also been analyzed with respect to aspects such as harmfulness/usefulness and maintenance issues of clones, taxonomies of clones or evolution studies of clones [11].

In this thesis, we examine more than twenty open source C, Java and C# systems, including the entire Linux Kernel, *j2sdk-Swing* and *db4o*, and compare their use of cloned code in several different dimensions, including language, clone size, clone similarity (i.e., with different UPI thresholds), clone location and clone density both by proportion of cloned functions and lines of cloned code [8, 12]. In particular, we focus on the following four research questions in this study:

(1) *Is NICAD capable of detecting clones from large systems? Can it detect clones from systems of different languages?* (2) *What is the cloning status of open source systems? Are there many clones? Are there more near-miss clones than exact clones?* (3) *Are there significant differences in cloning between the different language paradigms used in open source systems?* (4) *Are there significant differences in cloning in large systems compared to medium and small-sized open source systems?*

As regards the research questions, for the first question, we showed that NICAD is capable of accurately finding both exact and near-miss function clones even in large systems and different languages. For the second question, we can say that our results indicate a large number of exact function clones in these open source systems. We also see much higher percentages of near-miss clones, indicating significantly higher numbers of near-miss clones than exact clones in these systems. For the third question, we can say that we observed many more exact function clones in object-oriented Java and C# systems than in C systems. However, the effect of increasing the UPI threshold for near-miss clones was almost identical regardless of language paradigm. For the fourth question, we observed no significant differences in cloning related to the size of the systems. Even though Linux is huge, its cloning characteristics seem to be typical of other C systems. Similarly, the largest Java system *Swing* and the largest C# system *db4o* seem to be representative of the cloning characteristics of other systems written in their corresponding languages. The detailed results are available in an online repository [6] in a variety of formats and can be used as a benchmark.

8. Conclusion and Future Research Directions

Although clone detection is an active research area, there is a marked lack of work in the detection and analysis of near-miss software clones. In this thesis, we have advanced the state-of-the-art in clone detection research by building a hybrid clone detection method, proposing a metamodel of clone types, providing a scenario-based comparison of the clone detection techniques and tools, building a mutation-based framework for automatically evaluating clone detection tools and by a conducting large scale empirical study.

NICAD cannot detect *Type 4* semantic clones well. We thus plan to further hybridize NICAD with other techniques (e.g., Latent Semantic Indexing and Program Dependency Analysis) so that we can efficiently deal with semantic clones. We also plan to conduct a mega tool comparison experiment with third party tools using the mutation-based framework proposed in this thesis. We further plan to adapt NICAD in the development process. For example, one immediate plan is to build an Eclipse plug-in on top of NICAD that will assist developers in making clones (e.g., efficient and error-free copy/paste and renaming). A wide range of other future research ideas have been discussed in detailed in the original thesis [13] and in our technical report [11].

Acknowledgements: The author would like to thank Jim Cordy for his constant guidance and encouragement during the thesis work. This work is supported in part by NSERC and by an IBM Faculty Award.

References

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

[2] 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.

[3] J.R. Cordy. The TXL Source Transformation Language. *Science of Computer Programming* 61(3):190-210, 2006.

[4] E. Juergens, F. Deissenboeck, B. Hummel and S. Wagner. Do Code Clones Matter? In *ICSE*, pp. 485-495, 2009.

[5] C.K. Roy and J.R. Cordy. A Mutation / Injection-based Automatic Framework for Evaluating Clone Detection Tools. In *Mutation'09*, pp. 157-166, 2009.

[6] C.K. Roy and J.R. Cordy. JSME'09 Clone Results: `http://www.cs.queensu.ca/home/stl/download/NICADOutput/`.

[7] C.K. Roy, J.R. Cordy and R. Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. Special Issue on ICPC 2008, Science of Computer Programming, 74 (2009) 470-495, 2009.

[8] C.K. Roy and J.R. Cordy. Near-miss Function Clones in Open Source Software: An Empirical Study. Special issue on WCRE'08, submitted to *Journal of Software Maintenance and Evolution: Research and Practice*, 23pp., 2009 (in revision).

[9] 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.

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

[11] 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.

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

[13] C.K. Roy. Detection and Analysis of Near-miss Software Clones. Ph.D. Thesis, Queen's School of Computing, 263 pp., 2009 (submitted).