Towards a Mutation-Based Automatic Framework for Evaluating Code Clone Detection Tools

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

ABSTRACT

In the last decade, a great many code clone detection tools have been proposed. Such a large number of tools calls for a quantitative comparison, and there have been several attempts to empirically evaluate and compare many of the state-of-the-art tools. However, a recent study shows that there are several factors that could influence the the validity of the results of such comparisons. In order to overcome the effects of such factors (at least in part), in this student poster paper we outline a mutation-based controlled framework for evaluating clone detection tools using edit-based mutation operators that model cloning actions. While the framework is not yet completely implemented and as yet we do not have experimental data, we anticipate that such a framework will provide a useful contribution to the community by providing a more solid objective foundation for tool evaluation.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring, reverse engineering, and reengineering; D.2.5 [**Testing and Debugging**]: Testing tools

General Terms

Experimentation, Measurement

Keywords

Evaluation, Clone Detection Techniques, Mutation Analysis, Framework, Software Engineering, Maintenance.

1. INTRODUCTION

Copying a code fragment and reusing it by pasting with or without minor modifications is a common practice in software development, and 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 fraction (between 7% and 23%) of the code in a software system has been cloned [3]. While such cloning is often intentional [10] and not necessarily harmful [9], a difficulty introduced by such duplicated code is that when a bug is detected in a cloned code fragment, all fragments similar to it should be investigated for the same bug [12]. Moreover, when enhancing or adapting a software system, duplicated code can multiply the work to be done [8].

Fortunately, several (semi-)automated techniques for detecting code clones have been proposed (see [13, 14] for a comprehensive summary of all the available techniques) and there have been a number of comparison and evaluation studies to relate them [4, 6, 16, 5, 11] in different contexts. These studies have not only provided significant contributions to the clone detection research, but have also exposed how challenging it is to compare different tools, due to the diverse nature of the detection techniques, the lack of standard similarity definitions, the absence of benchmarks, the diversity of target languages, and the sensitivity to tuning parameters [2].

The most recent study, by Bellon et al. [4], provides a comprehensive quantitative evaluation of six clone detectors in detecting known observed clones in a number of open source software systems written in C and Java. However, even in that careful study, only a small proportion of the clones were oracled, and a number of other factors have been identified as potentially influencing the results [2]. The general lack of evaluation is exacerbated by the fact that there are no agreed upon evaluation criteria or representative benchmarks. Finding such universal criteria is difficult, since techniques are often designed for different purposes and each has its own tunable parameters.

In an attempt to compare all clone detection techniques more uniformly, independent of tool availability, implementation limitations or language, in an another study we have taken a predictive, scenario-based approach [14]. We have designed a small set of hypothetical program editing scenarios representative of typical changes to copy/pasted code. Based on these hypothetical scenarios, we have estimated how well the various clone detection techniques may perform based on their *published* properties. In order to estimate maximal potential, we have also assumed the most lenient settings of any tunable parameters of the techniques. Thus, our that study is not an actual evaluation, but rather an attempt at an overall picture of the potential of the techniques in handling clones resulting from each of the scenarios.

In order to automatically compare and evaluate the techniques and tools in a more realistic setting, in this student poster paper we have outlined a mutation-based evaluation

C3S2E-08 2008 May 12-13, Montreal [QC, CANADA]

framework which is still under implementation. Mutation analysis has been used in the testing community for over 25 years and has proven to be a useful comparison metric for assessing and improving multiple test suites [1]. We believe that a mutation-based testing approach can also play an important role in evaluating clone detection tools, if we can define a set of mutation operators that model cloning for a given language. Thus, in this paper, we begin by outlining an editing taxonomy for code cloning from which a set of mutation operators for cloning can be derived, and then propose an evaluation framework based on these operators that can be used to evaluate and compare clone detection tools using clone mutation.

The rest of this paper is organized as follows. After introducing mutation operators for cloning in the C language in Section 2, we outline our proposed framework in Section 3. In Section 4 we provide a brief discussion of existing evaluation experiments and finally, Section 5 concludes the paper.

2. MUTATION OPERATORS FOR CLONING

For any mutation-based analysis, availability of a set of representative mutation operators is a primary concern. While there are numerous mutation generators available for generating potential "bugs" in various languages [1], mutation operators for code cloning have to our knowledge not been studied so far. In this section, we attempt to provide an editing taxonomy of different code clone types from which a set of mutation operators for code cloning can be derived to support the mutation analysis of the next section.

In mutation testing analysis, mutation operators are targeted to change the original code so as to introduce new potential bugs. Similarly, mutation operators for cloning are those editing activities that create new clones when applied to copy/pasted code. In our earlier study [14], we have proposed 16 different (sub-)types of clones by refining the available clone types from the literature [13]. Each of the 16 clone types (e.g., changes in format, renaming of identifiers, etc.) potentially reflect 16 kinds of mutation operators for cloning (Our another recent study [15] shows that there are actually many such clone types in real systems).

However, in that study only single-level editing operations are applied to the copied code. In practice, a copied code fragment may undergo several kinds of editing. Consider the editing taxonomy of Figure 1 where the original segment is marked with *Original Fragment*. While the top-right code segment has undergone only one kind of editing (a formatting change), the bottom-right segment has undergone several different kinds of editing operations. If we follow the solid arrows in the taxonomy from the original fragment, we can determine the editing operations applied to that segment (Formatting change => systematic renaming of identifiers => expressions for parameters => small insertion within a line => insertion of new lines => reordering of some statements => control replacement). Thus, there should be clone mutation operators not only for each of the editing operations, but also for combinations of them. We are using TXL [7] for generating the mutation operators as a sequence of source transformations to allow for this.

3. PROPOSED FRAMEWORK

The conceptual diagram of our proposed mutation-based framework is shown in Figure 2. While this framework is mutation-based and follows the basic principle of mutation analysis, there are several additional factors adapted and added to make it suitable for clone detection analysis. In the following we discuss the different components/phases of the framework in brief.

Select Code Base: Selecting the target code base is the first step of the framework. One needs to determine which code bases would be meaningful for such an evaluation in terms of subject tools and the size of the code. In the first instance, we plan to use the four C systems of the Bellon et al. [4] study. C is supported by most of the clone detection tools, and we are targeting our clone mutation operators at C systems at that moment. The size of these systems is also reasonable for the framework.

Extract Exact Clones: After selecting a subject code base, one needs to detect all the exact clones of that system using any good clone detection technique. The detected clones may be manually verified to gain confidence in the identified clones, although fortunately most tools detect exact clones with few or no false positives.

Annotate Exact Clones: After identifying the exact clones in the code base, a TXL [7] source transformation is used to annotate the exact clone pairs/classes in the original code. The output of this phase is the annotated code base where annotations mark the clone relation between the clones pairs/classes.

Generate Clone Mutants: A TXL-based Mutation Generator (cf. Section 2) is then randomly applied to the annotated code base. Mutation is only applied to one of the annotated code segments at a time and one mutant version of the code base is created (say mutant 1). In this way, thousands of mutant versions of the annotated code base are created by applying the mutation operators for cloning. Each mutated clone pair/class is marked and stored in a database with the identity of the applied mutation operator for later analysis.

Apply the Subject Tools to the Mutant Versions: All the subject clone detection tools are used to detect clones (possibly with different settings of the different tools) in each of the mutant versions.

Determine If Mutated Clone Pair/Class is Detected or Not: For each of the tools and each of the mutant versions, the framework will determine whether the mutated clone pair/class has been detected by the corresponding tool. If the mutated clone pair/class is detected, we say that the tool has performed well for the clone type represented by the mutation operator.

Analyze and Display Results: Once all the n different mutant versions of the code base are processed by all the k subject clone detection tools, a comparison of the results for the different mutation operators (hence, for the different clone types) for different tools is displayed to the user. While in this first instance these results will be based on only one subject system per run, it is of course possible to provide more comprehensive results by observing several different subject systems.

4. RELATED WORK

Although as yet there is no mutation-based evaluation framework available, there are several experiments that compare and evaluate clone detection tools/techniques. In this section, we provide a summary of the available tool comparison experiments from the literature.

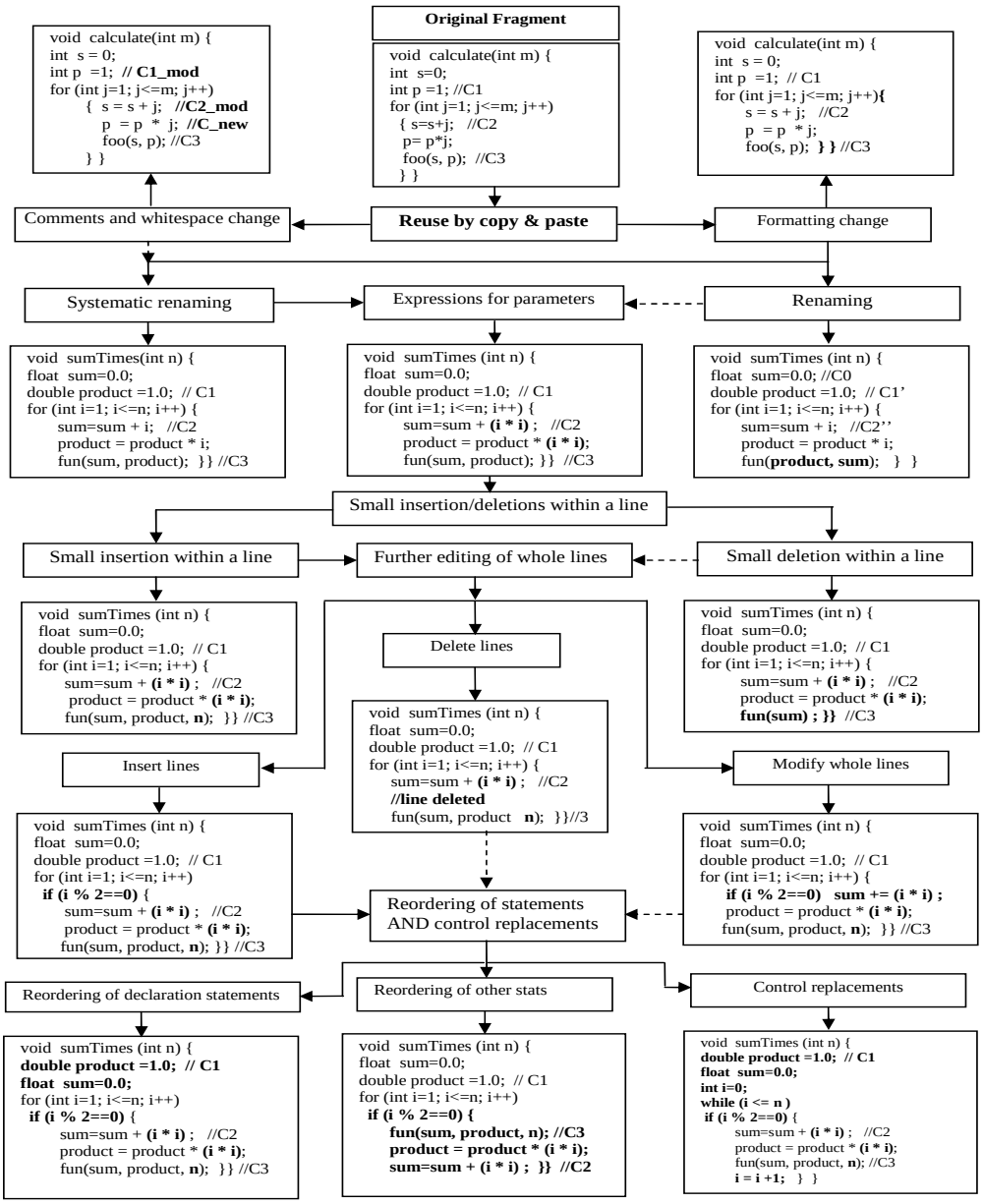

Figure 1: Taxonomy of Editing Scenarios for Different Clone Types

One of the first experiments was conducted by Bailey and Burd [6] where they compared three state-of-the-art clone detection and two plagiarism detection tools. First they validated all the clone candidates of the subject application obtained with all the techniques of their experiment. A human oracle was then used to compare the different techniques in terms of several metrics to measure various aspects of the found clones.

Although they were able to verify all the clone candidates, the limitations of the case study in terms of system quantity and size makes their findings questionable. Moreover, the intention of their analysis was to assist in preventative maintenance tasks, which may also have had an influence in validating the candidate clones.

Considering the limitations of Burd and Bailey's study, Bellon et al. conducted a larger tool comparison experiment [4] with the same three clone detection tools used in Burd and Bailey's study and three additional clone detection tools. They also used a more diverse set of software systems, 4 Java and 4 C systems totalling almost 850KLOC. As in the study of Burd and Bailey, a human oracle validated the candidate clones from all the tools. While their study is the most extensive to-date, only a small proportion of the clone candidates were oracled and several other factors might have influenced the results [2]. Although this study was later extended by Koschke et al. [11] with prototype implementations of several tools, they did not address anything to overcome the limitations of Bellon et al.'s study.

Rysselberghe and Demeyer [16] evaluate three representative clone detection techniques and provide comparative results in terms of portability, kinds of duplication reported, scalability, number of false matches, and number of useless matches. However, they have used small/medium size (under 10KLOC) cases and prototype implementations of the

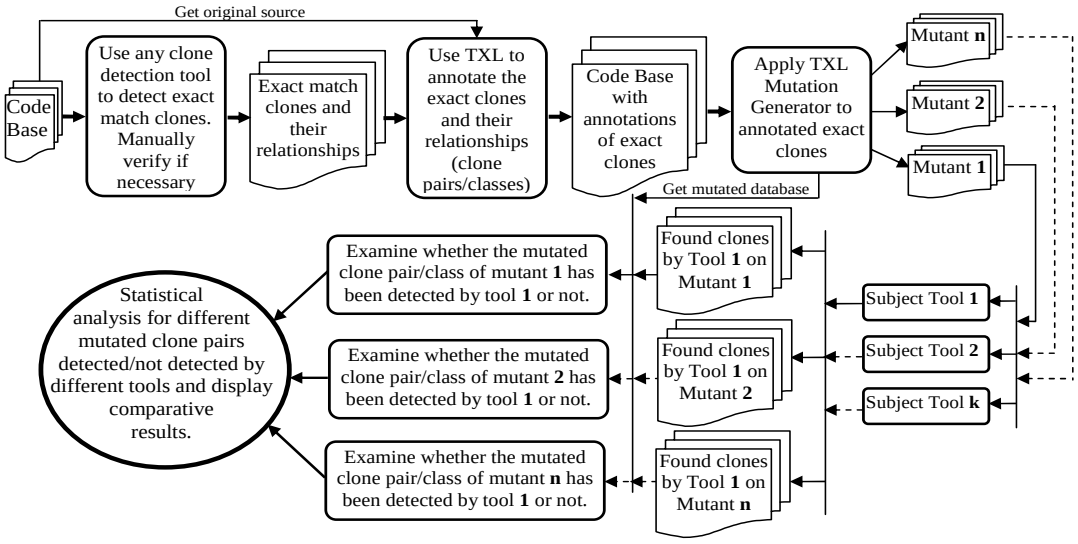

Figure 2: The Proposed Mutation-Based Evaluation Framework

tools instead of using the actual tools. Moreover, rather than quantitative evaluation of the detection techniques, their intention was to determine the suitability of the clone detection techniques for a particular maintenance task (e.g., refactoring).

Another interesting study has been conducted by Bruntink et al. [5] where several clone detection techniques are evaluated in terms of finding cross-cutting concerns in C programs with homogeneous implementations.

5. CONCLUSION

Existing studies for empirically evaluating clone detection tools have had several limitations and thus, cannot provide a convincing comparative study. In this student poster paper, we propose a new approach for evaluating clone detecting tools in a controlled way by borrowing an established technique from the testing community – mutation-based analysis. Although we have not yet completed the implementation of the framework, we are confident that such a framework can provide concrete and accurate comparative results for different tools in finding intentionally created code clones. In this proposed framework, it is not practical to work with large scale code bases (as thousands of different mutated versions of the code base are fed into the clone detection tools). Thus, in future we also plan to conduct another mutation/injection-based controlled experiment. In that framework, thousands of mutated clone pairs/classes generated from the editing taxonomy (Figure 1) will be injected to large systems. Clone detection tools will then be evaluated how well and how fast they can detect the known injected clones.

6. ACKNOWLEDGEMENTS

The authors would like to thank the three anonymous reviewers for their valuable comments in improving the paper. This work is supported by the Natural Sciences and Engineering Research Council of Canada.

7. REFERENCES

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

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

[3] B. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *WCRE*, pp. 86-95, 1995.

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

[5] M. Bruntink, A. Deursen, R. Engelen and T. Tourwe. On the Use of Clone Detection for Identifying Crosscutting Concern Code. *IEEE TSE*, 31(10): 804-818, 2005.

[6] E. Burd and J. Bailey. Evaluating Clone Detection Tools for Use during Preventative Maintenance. In *SCAM*, pp. 36-43, 2002.

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

[8] J. Johnson. Visualizing Textual Redundancy in Legacy Source. In *CASCON*, pp. 171-183, 1994.

[9] C. Kapser and M. Godfrey. "Cloning Considered Harmful" Considered Harmful. In *WCRE*, pp. 19-28, 2006.

[10] M. Kim and G. Murphy. An Empirical Study of Code Clone Genealogies. In *FSE*, pp. 187-196, 2005.

[11] R. Koschke, R. Falke and P. Frenzel. Clone Detection Using Abstract Syntax Suffix Trees. In *WCRE*, pp. 253-262, 2006.

[12] 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.

[13] 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.

[14] C.K. Roy and J.R. Cordy. Scenario-Based Comparison of Clone Detection Techniques. In *ICPC*, 10 pp., 2008 (to appear).

[15] C.K. Roy and J.R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *ICPC*, 10 pp., 2008 (to appear).

[16] F.V. Rysselberghe and S. Demeyer. Evaluating Clone Detection Techniques. In *ELISA*, 12pp., 2003.