A Mutation Analysis Based Benchmarking Framework for Clone Detectors

Jeffrey Svajlenko* Chanchal K. Roy* James R. Cordy+

*Department of Computer Science
University of Saskatchewan
Saskatoon, Canada
{jeff.svajlenko, chanchal.roy}@usask.ca

+School of Computing
Queen's University
Kingston, Canada
cordy@cs.queensu.ca

***Abstract*—In recent years, an abundant number of clone detectors have been proposed in literature. However, most of the tool papers have lacked a solid performance evaluation of the subject tools. This is due both to the lack of an available and reliable benchmark, and the manual efforts required to hand check a large number of candidate clones. In this tool demonstration paper we show how a mutation analysis based benchmarking framework can be used by developers and researchers to evaluate clone detection tools at a fine granularity with minimal effort.**

***Index Terms*—Evaluation, Benchmarking, Clone Detection, Mutation Analysis, Framework.**

I. Introduction

The performance of clone detection tools is measured in terms of recall (% of true clones within a system found) and precision (% of clones reported which are true clones). Measuring precision involves validating a tool's complete (or partial) output. Much more difficult is measuring recall as knowledge of the true clones within the system must be known. Recall is therefore typically measured by comparing the tool's output to a clone corpus built for the system. Previously corpuses have been constructed by consulting various clone detection tools, and validating a random selection of their output [1]. The difficulty with this technique is the amount of manual effort required to validate a large corpus. Additionally, the corpus becomes specific to and biased by the tools used to construct it. The tool we present here resolves these two difficulties by synthesizing a corpus rather than mining for one. Synthesis can be automated and controlled which allows a well understood and unbiased corpus to be created with almost no human effort.

II. Framework

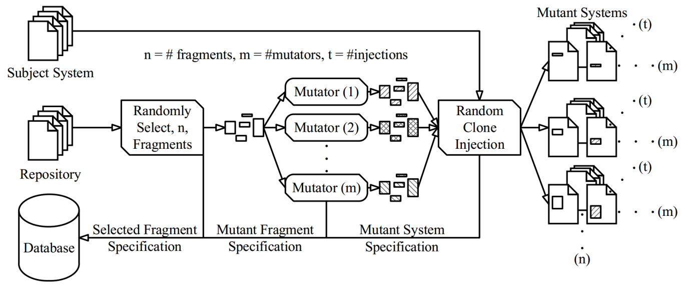
This framework has two phases. In the first phase (generation) a corpus of clones is synthesized and then hidden in a *subject system* for the tools to search. The second phase (evaluation) executes the detection tools for the corpus and analyzes their output to measure recall and precision of the synthesized clones (only).

Clone Synthesis. The framework synthesizes clones by mutating real code fragments mined from a *source code repository*. The original and mutated fragments form a clone pair produced by mimicking copy, paste and modify cloning behavior. Mutations are based upon a validated comprehensive taxonomy of the types of edits developers usually perform on copy and pasted code [4]. The framework performs mutation using *mutation operators* (Table 1) which takes a fragment as input and outputs the same fragment with a single random edit of the defined edit type. Framework users specify the types of clones to generate as *mutators*, which are sequences of mutation operators which are applied one by one to an input fragment.

Generation Phase. This phase (Fig. 1) begins by selecting and extracting any given number *n* of existing code fragments from a code repository. These fragments are mutated by *m* user-defined mutators. The resulting *mutant fragments* are paired with their *selected fragment* to form a corpus of *nm* synthesized clones. For each of these clones, a mutated version of the subject system is created by injecting the selected and mutant fragments into the subject system at random syntactically correct locations. These *mutant systems* evolve the original subject system by a copy-paste-modify cloning activity, and contain exactly one clone from the generated clone corpus. The framework can be configured to produce multiple mutant systems per generated clone pair using different injection locations. The framework tracks corpus generation details in a database.

TABLE I. Mutation Operators From Editing Taxonomy

Name	Mutation Description	Clone Type
mCW_A	Change in whitespace (addition).	1
mCW_R	Change in whitespace (removal).	1
mCC_BT	Change in between token (/* */) comments.	1
mCC_EOL	Change in end of line (//) comments.	1
mCF_A	Change in formatting (addition of newlines).	1
mCF_R	Change in formatting (removal of newlines).	1
mSRI	Systematic renaming of an identifier.	2
mARI	Arbitrary renaming of a single identifier.	2
mRL_N	Change in value of a single numeric literal.	2
mRL_S	Change in value of a single string literal.	2
mSIL	Small insertion within a line.	3
mSDL	Small deletion within a line.	3
mILs	Insertion of a line.	3
mDLs	Deletion of a line.	3
mMLs	Modification of a whole line.	3



Subject System
Repository
Database
n = # fragments, m = #mutators, t = #injections
Randomly Select, n, Fragments
Mutator (1)
Mutator (2)
Mutator (m)
Random Clone Injection
Mutant Systems
(t)
(m)
(n)
Selected Fragment Specification
Mutant Fragment Specification
Mutant System Specification

Fig. 1. Generation Phase

The framework allows a number of constraints to be placed on the generated clone corpus, including: min/max clone size (lines/tokens), minimum clone similarity, and mutation position (min. distance from start/end). This last constraint is to encourage the tools to detect the mutant portions (especially for type 3 clones) and not trim them into multiple identical clones.

Evaluation Phase. The evaluation phase (Fig. 2) proceeds by running the tools for each of the mutant systems. Any tool can be added to the framework by implementing a tool runner, a simple communication protocol between the framework and the tool. The tool's performance is measured per mutant system as *unit recall* and *unit precision*. The framework tracks tool unit performance in a database.

Unit recall is 1.0 if a tool detects the injected clone, or 0.0 if it does not. Successful detection is determined by a subsume clone matching algorithm with a parameterized tolerance threshold. Framework users may also specify a required matching clone similarity threshold to prevent false positives subsuming the clone from being considered a successful match.

Unit precision is measured by validating additional clones found by the tool that contain at least one of the injected fragments. This measures the precision impact of the injected clone. Validation is mostly automated using a clone validator, which uses source code normalization and multiple code similarity metrics to make its decision, as well as its knowledge of the types of clones created. Minimal human validation is required when the validator is unconfident in judging a clone.

Output: Once the unit performances have been evaluated, overall performance is found by averaging the unit performances. This allows recall and precision to be reported at various granularities, including: per clone type, per user-defined mutator, and per mutation operator.

Scope: The framework supports function and block level clones of the Java, C and C# programming languages.

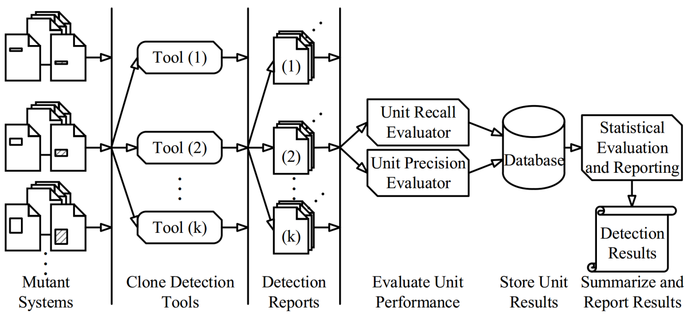


Fig. 2. Evaluation Phase

III. Previous Work

Previous work includes introduction of the framework's methodology and a prototype experiment [3]. The prototype was developed to specifically target NiCad [2] variants, which greatly simplified the underlying algorithms. This work has generalized the framework for use with any clone detection tool, which required significant redesign and reimplementation of the underlying algorithms. The generation phase has been improved to provide more customization and control over corpus generation, and expands language and granularity support. The evaluation phase has also been improved, including a new automatic clone validator.

IV. Usability

The framework is operated by a simple menu-based command line interface. A menu-based interface was chosen as it allows us to document all options within the application, greatly simplifying its operation. All configuration parameters are explained and provide guidance for selection, including defaults. By presenting the framework at the command line, it can be easily executed remotely (e.g., cloud computing).

The menu lists context sensitive options for the six stages of the experiment, which include: (1) experiment creation, (2) generation phase configuration, (3) generation phase, (4) evaluation phase configuration, (5) evaluation phase, and (6) result summary and review. It is possible to re-execute the evaluation phase by returning to stage (4) from stage (6). Any evaluation phase configuration (including participating tools) may be changed. Subsequent executions of the evaluation phase will reuse any detection or evaluation data not affected by the configuration change.

Between any of these stages the experiment can be exited and later resumed. Experiments and their data are portable, allowing for easy export and import. Our intention is for standardized datasets to be generated and shared amongst the clone community. A standard benchmark allows tools to be compared without requiring the tools be evaluated together.

V. Demonstration

In this tool demonstration, we will show how the framework can be used both to evaluate individual detection tools and to compare a set of subject tools.

References

[1] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Transactions on Software Engineering*, 2007, pp. 577-591.

[2] J.R. Cordy and C.K.Roy, 2010, "The NiCad Clone Detector," in *Proc. of the Tool Demo Track of the 19th International Conference on Program Comprehension*, 2011, pp. 219-220.

[3] C.K. Roy and J.R. Cordy, 2009. "A Mutation / Injection-based Automatic Framework for Evaluating Code Clone Detection Tools," in *Proc. of the ICST 4th Int. Workshop on Mutation Analysis*, 2009, pp. 157-166.

[4] C.K. Roy, J.R. Cordy and R. Koschke, 2009, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach," *Science of Computer Programming*, 74 2009, pp. 470-495.