A Mutation / Injection-based Automatic Framework for Evaluating Code Clone Detection Tools

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

Abstract

In recent years many methods and tools for software clone detection have been proposed. While some work has been done on assessing and comparing performance of these tools, very little empirical evaluation has been done. 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 manual effort required to hand check large numbers of candidate clones. In this paper we propose an automated method for empirically evaluating clone detection tools that leverages mutation-based techniques to overcome these limitations by automatically synthesizing large numbers of known clones based on an editing theory of clone creation. Our framework is effective in measuring recall and precision of clone detection tools for various types of fine-grained clones in real systems without manual intervention.

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 [5, 41]. While such cloning is often intentional [28] and not necessarily harmful [3, 25], there is a consensus that clones should be detected, both to overcome the maintenance difficulties posed by such clones, and to address a range of other tasks that require extraction of similar code fragments [39]. In response, over the past decade many tools and techniques for detecting code clones have been proposed. Some of these are text-based [16, 23, 35, 37], some are token-based [5, 24, 34], some are tree-based [8, 17, 18, 22], some are metrics-based [31, 30, 36] and some are graph-based [19, 29, 33].

Given the similarity of clone detection to information retrieval (IR) [31], precision and recall are important factors in measuring and comparing the effectiveness of these tools, especially when dealing with large systems. Unfortunately, despite a decade of active research, there has been a marked lack of in-depth evaluation in terms of precision and recall. Although in some cases precision has been measured by manually validating a very small set of randomly selected clone candidates [19, 22, 29, 33, 34], recall has not been measured so far, in part due to the practical difficulties in measuring it. While tool comparison experiments [9, 10, 13, 42] and some individual tool evaluations [18, 31, 37] have been designed to measure precision and recall (as well as time and space), these efforts have faced serious challenges with the difficulty of creating a large enough base of reliable reference data and the expense of manually validating thousands of candidate clones.

As a result, these experiments have either used no reference data [42], have simply assumed that the union of all tool results from a set of subject systems was representative (possibly hand validating a small sample) [9, 10, 13, 18] or have manually validated clones detected in a small subject system using a human oracle [12, 31, 37]. While the union reference data may give good relative measurements [10], one cannot guarantee that the subject tools have indeed detected all the clones from the subject systems. Manually validating the large number of candidate clones or manually oracling a subject system is also a real challenge. For example, in Bellon's experiment [10], it took 77 hours for Bellon to validate only 2% of the candidate clones. While oracling small systems is possible [13, 31, 37], even a relatively small system such as *Cook*, when looking only for function clones, has nearly a million function pairs to sort through [43], an impossible task to be handled error-free by a human. Moreover, none of the experiments or individual authors report on the reliability of the human judges, and even expert judges frequently disagree in creating task-relavent reference data [43]. Furthermore, with the exception of Bellon et al. [10] (for clone types 1, 2 and 3), Falke et al. [18] (for clone types 1 and 2) and a hypothetical evalu-

ation by Roy and Cordy [38] there is no work that separates precision and recall values for different types of clones.

In this paper, we propose a mutation-based [2] approach that automatically and efficiently measures (and compares) the recall and precision of clone detection tools for different types of fine-grained copy/paste/modify clones. We propose a taxonomy of clone types that reflects a developer's typical copy/paste editing activities, and design code mutation operators to model each type. By using these operators to generate and track a large number of artificial clones, we then automatically measure how well (i.e., precision) and how efficiently (i.e., recall) these known clones are detected by a particular tool (for individual tool evaluation) or group of tools (for comparing different tools).

The rest of this paper is organized as follows. Following a short introduction to terminology and general types of clones in Section 2, we present our proposed editing taxonomy of fine-grained clone types in Section 3. We present our corresponding mutation operators in Section 4, and the details of the mutation-based evaluation framework in Section 5. In Section 6 we report the results of an example use of the framework. While in Section 7 we consider existing evaluation studies and their relation to ours, Section 8 concludes the paper with our plans for future research.

2. Background

We begin with a basic introduction to clone detection terminology.

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, sequence of statements, or so on. A CF is identified by its file name and begin-end line numbers in the original code base and is denoted as a triple *(CF.FileName, CF.BeginLine, CF.EndLine)*.

Definition 2: Code 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) = CF2* where *f* is the similarity function (see *clone types* below). Two fragments that are similar to each other form a *clone pair* (e.g., *(CF1,CF2)*), and when many fragments are similar, they form a *clone class* or *clone group*.

Definition 3: Clone Types. There are two main kinds of similarity between code fragments. Fragments can be similar based on the similarity of their program text, or they can be similar base on in their functionality (independent of their text). The first kind of clone is often the result of copying a code fragment and pasting into another location.

- Textual Similarity: Based on the textual similarity we distinguish the following types of clones [10]:

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

  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.

- Functional Similarity: If the functionality of two code fragments is identical or similar, we call them *semantic* or *Type 4* clones [19, 29].

  Type 4: Two or more code fragments that perform the same computation but are implemented by different syntactic variants.

3. An Editing Taxonomy for Cloning

The definition of clone in the literature is inherently vague. Baxter et al. [8] give the most general definition, defining clones simply as segments of code that are similar according to some definition of similarity. Kamiya et al. [24] define clones as portions of source file(s) that are "identical" or "similar" to each other, where by identical they mean exact copy, but similar is undefined, and a similar definition is used by Burd et al. [13], where a code segment is termed a clone if there are two or more occurrences of the segment in the source code with or without "minor" modifications, where minor is undefined. Several authors, including Baxter et al. [8], have defined "similar" using detection-dependent definitions in terms of difference thresholds [26, 31, 34], and it has been proposed that automatically combining multiple detector result sets can help overcome such similarity definition problems [10, 31]. Categorization in the form of clone taxonomies has been suggested as a way to avoid such ambiguities in definition [6, 36]. However, these taxonomies are limited to function clones and still use vague terms such as "similar" [36] and "one/two/three long difference" [6].

While each of these approaches may help in evaluating the tools in question, they still leave open the question of how well the results might match what human judges would decide, and make it difficult to compare methods on an objective basis. What we need is an evaluation system that can be used independently of detection method, and that demonstrably matches human judgement well. Given the problems with validating existing clones, we propose instead to use a theory of clone creation to mimic the actions of a development programmer in synthesizing new clones that can be used to objectively evaluate detection methods with no need for hand validation.

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

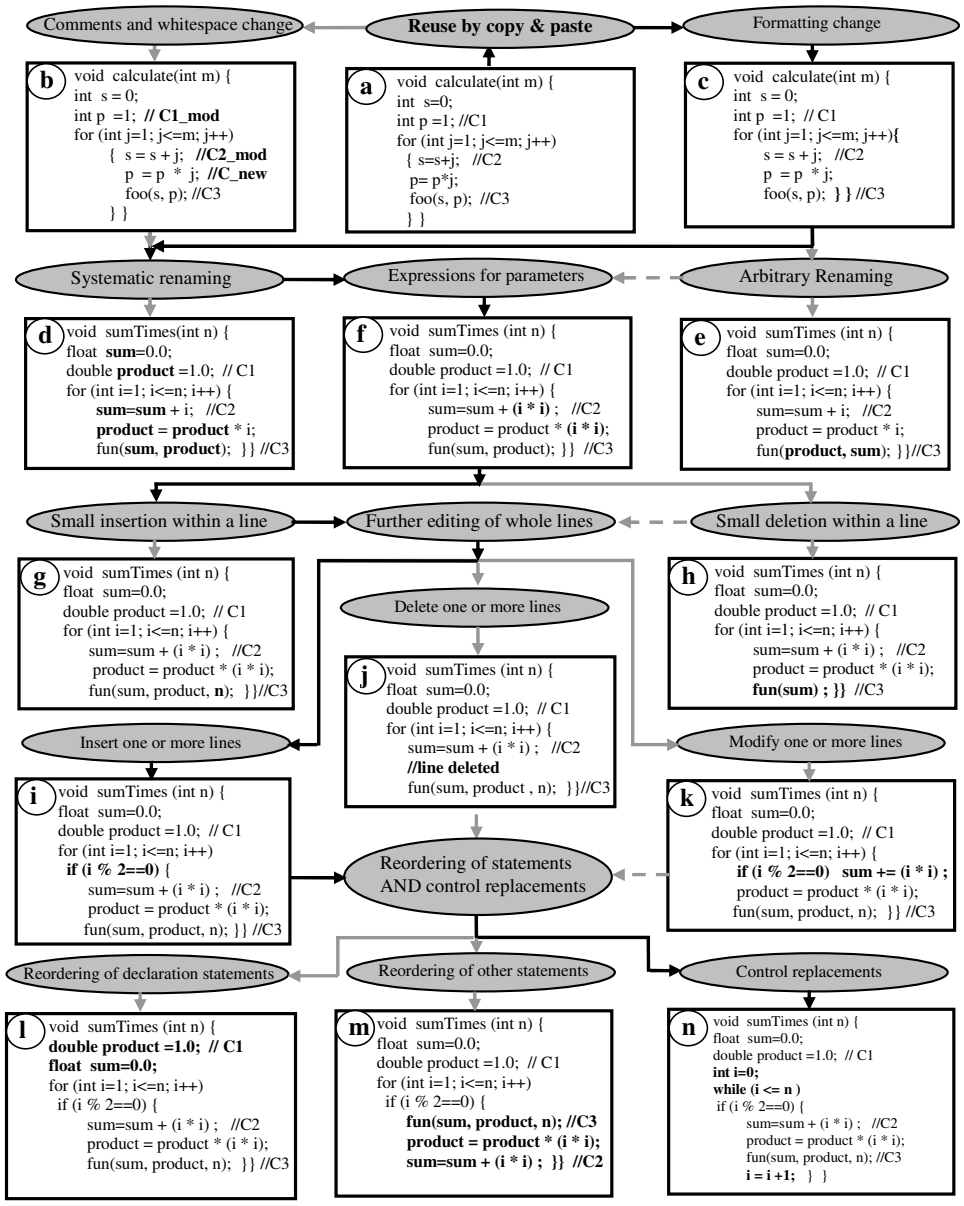


Figure 1. Example Application of the Editing Taxonomy for Cloning

we have formalized into a taxonomy of the editing actions that a programmer may undertake in the intentional creation of a clone [38]. Our taxonomy is not simply guesswork - it is derived from the large body of published work on existing clone definitions [8, 19, 24, 29, 34], clone types [10, 31], clone taxonomies [6, 26, 36], studies of developer copy/paste activities [27] and other empirical studies [3, 7, 25, 28]. We have validated the taxonomy by studying the copy/paste patterns of function clones in an empirical study that analyzed 17 open source C and Java systems including the entire Linux Kernel (6265 KLOC C, 154977 functions), Apache *httpd* (275 KLOC C, 4301 functions) and *j2sdk-swing* (204 KLOC Java, 10971 methods) [41].

Figure 1 demonstrates the use of our proposed editing taxonomy for code fragments at the function-level of granularity. The taxonomy is demonstrated on a simple example original function (a) that calculates the sum and product of a loop variable and calls another function with these values as parameters. Although the editing steps are demonstrated at function-level granularity, they are general enough to be applicable to any granularity of code fragment.

While we claim that the proposed taxonomy is a comprehensive one, we cannot guarantee that clones created by this editing taxonomy are representative for any particular task, such as software maintenance by refactoring. What we can guarantee with some confidence is that there are no

clone types in the literature that cannot be created using our taxonomy. (For full details on different clone types and taxonomies with concrete examples, the reader is referred to our recent technical report [39]).

4. Mutation Operators for Cloning

Although mutation has been used primarily in the testing community, we have previously applied mutation-based analysis to the comparative evaluation of fault detection techniques for concurrent software [11], an area where objective evaluation is notoriously difficult. With this previous experience and in light of the similar difficulties in evaluating clone detection techniques, we use mutation as the basis of our clone detection tool evaluation framework.

While mutation operators for generating potential "bugs" in various languages have been extensively studied, mutation operators for code cloning are to our knowledge unique to our work. In mutation testing analysis, mutation operators are targeted to change the original code so as to introduce new potential bugs. Analogously, mutation operators for cloning should create new clones by modeling developers' copy/paste editing activities, exactly what our editing taxonomy (Section 3) encodes.

In this section we outline the set of mutation operators that we have developed using the TXL source transformation language [14] for generating each of the different types of clones (and combinations thereof) in Section 3 given an original code fragment. Table 1 summarizes the set of mutation operators we have developed and used in our evaluation framework (Section 5). The first column of the table gives a name to the mutation operator, the second lists the associated editing activities, the third provides an example by reference to the fragments of Figure 1, and the fourth gives the clone type of the mutated result.

Like the mutation operators used in mutation testing, each of our cloning operators is targeted to make just one random change. For example, the mutation operator *mCR* randomly replaces a single *for-loop* with an equivalent *while-loop*, for example producing fragment (n) as a mutant of fragment (i) in Figure 1. However, unlike mutation testing, we are interested in clones that can be created by sequences of editing operations, and thus sequences of our mutation operators can be applied to produce other clones, possibly of mixed type, as for example in the sequence of edits that leads from the original fragment (a) to the mutated fragment (n) in Figure 1 (solid black arrows). By applying the operators in random sequences to a code fragment, we can create hundreds of different cloned versions on which to test clone detectors. As in all mutation-based analysis, we cannot be certain that the created mutants actually reflect a developer's editing activities, but we know that they are similar.

Almost all of our mutation operators are language-independent, and can be applied to fragments of any granularity (e.g., functions, begin-end blocks, statement sequences, and so on). Only the keywords of the target language need be provided, so that they are not mistaken for identifiers by the mutator. The operator *mCR* is of course language-specific, and currently supports C, Java and C#, although adaptation to other languages is not difficult.

Table 1. Mutation Operators for Cloning

Name	Random Editing Activities	Ref**	Type
mCW	Changes in whitespace and	a ⇒ b	Type 1
mCC	Changes in comments		
mCF	Changes in formating	a ⇒ c	
mSRI	Systematic renaming of identifiers*	c ⇒ d	Type 2
mARI	Arbitrary renaming of identifiers*	c ⇒ e	
mRPE	Replacement of identifiers with expressions (systematically or non-systematically)	d ⇒ f	
mSIL	Small insertions within a line	f ⇒ g	Type 3
mSDL	Small deletions within a line	f ⇒ h	
mILs	Insertions of one or more lines	g ⇒ i	
mDLs	Deletions of one or more lines	g ⇒ j	
mMLs	Modifications of whole line(s)	g ⇒ k	
mRDS	Reordering of declarations	i ⇒ l	Type 4
mROS	Reordering of other statements+	i ⇒ m	
mCR	Replacing one type of control by another	i ⇒ n	

**Refers to the taxonomy (Figure 1) for the corresponding mutator
*Function names, variables, data types and literal values
+Data-dependent or independent statements

5. The Evaluation Framework

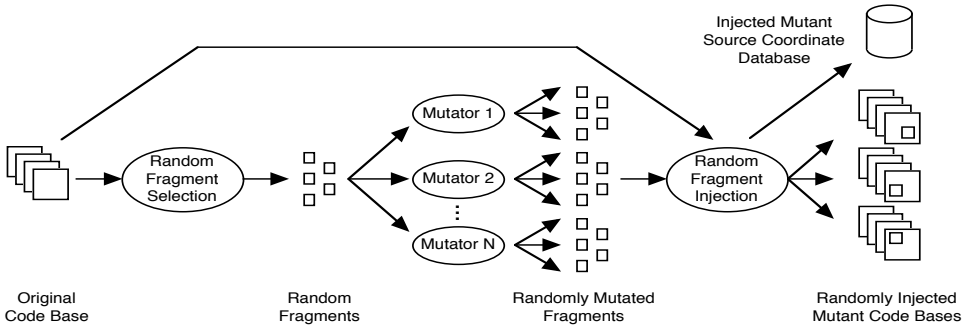
In this section, we provide the details of our evaluation framework. We begin with the components of the framework and then discuss evaluation strategies, language dependency, scalability and adaptability to other tools.

Figure 2 shows a conceptual diagram of our framework. The framework has two main phases, the *Generation Phase*, in which randomly mutated clone fragments are generated from the original code base and randomly injected into the code base to get mutated code bases, and the *Evaluation Phase*, in which the mutated code bases are used to evaluate and compare clone detection tools. The random mutation and injection steps allow for thousands of randomly placed clones to be generated.

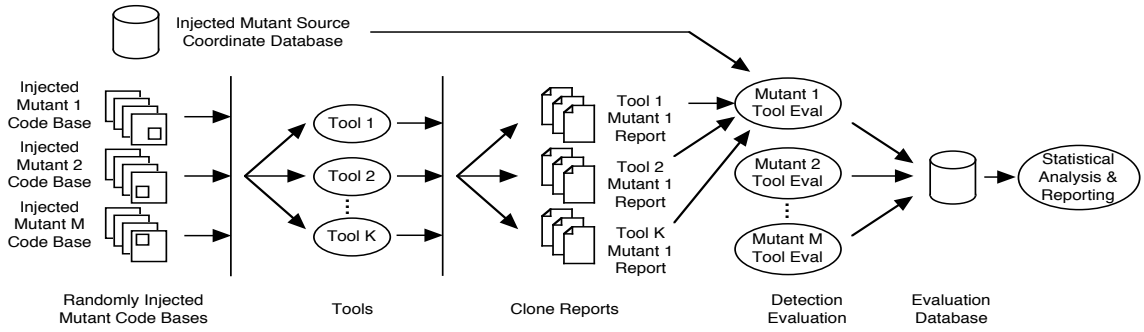
5.1 Clone Generation Phase

In the first phase (Figure 2(a)), any number of mutated versions of the code base containing injected mutant clones is created. Beginning with a subject system (code base) in a language supported by the tools under evaluation, the generation phase has the following steps.

Random Fragment Selection: Once the subject code base is selected, any desired number of existing code fragments from the code base are automatically and randomly



(a) Framework: Generation Phase



(b) Framework: Evaluation Phase

Figure 2. The Proposed Evaluation Framework

selected for clone mutation. Code fragments can be of any chosen granularity (e.g, function definition, begin-end block, statement sequence) and can be selected by hand if desired. Selected code fragments are stored as fragment files in a directory for further processing.

Mutant Clone Generation: A TXL-based mutation process then uses the clone mutation operators (Section 4) to mutate the selected code fragment files either randomly or sequentially to create a set of mutant clone fragments, which are stored as files in a second directory. Any number of mutant clone fragments can be generated from each selected fragment, using any fixed or random combination of the clone mutation operators.

Random Clone Injection: For each of the mutated code fragments, a mutated version of the code base is created by injecting the mutated code fragment at a random position in a randomly selected source file of the code base. (This contrasts with our original framework plan [40] in which we proposed to mutate known clones in place.) Random file and location injection mimics a developer's possible copy/paste scenario, yielding a set of realistic new code bases each with exactly one new copy/pasted clone. Information for each new code base with the location of its injected mutant clone, the mutation operators that were used to generate it, and the location of the original code fragment from which is was mutated is stored in a database for use in the evaluation phase (Figure 2(b)).

5.2 Tool Evaluation Phase

In the evaluation phase (Figure 2(b)) each of the mutant code bases is fed to each of the subject clone detection tools for evaluation and comparison. If the tool detects and pairs the injected mutant clone with its original fragment, then it is considered to have "killed" the mutant. This phase consists of the following major components:

Tool Runs: Each of the subject tools is run with each of the mutant code bases as input. Depending on the mutation operators used, each tool is run with its tunable parameters (if any) set to target the clone types of the injected mutants.

Detection Evaluation: For each tool with each mutant code base, the generated clone report is analyzed for unit recall (Section 5.3) and precision (Section 5.4) of the tool in detecting the injected mutant fragment and its original as a clone pair. Location and type of the mutant clone is provided by the database generated in the generation phase.

Evaluation Database: As the tools are run, unit recall and precision information for each mutant is added to an evaluation database for summary analysis. Information from the clone validator (see Section 5.6) is added to the database before analysis.

Statistical Analysis and Reporting: Once the experiment is completed, the evaluation database is used to calculate the observed summary recall and precision of each subject tool for each fine-grained clone type (i.e., individual mutation operator), each clone type (types 1 through 4),

and overall recall and precision. Accurately summarizing precision may involve a small amount of manual analysis for some clone pairs (see Section 5.6).

5.3 Measurement of Recall

One of the primary objectives of our work is to automatically measure the recall of clone detection tools. Our framework reports recall for each fine-grained clone type (i.e., individual mutation operator), each clone type (types 1 through 4), and overall for each tool. Our recall definition is the usual one in IR research, that is, the number of items detected divided by the total number of detectable items. In our case the detectable items are our injected mutant clones, and our mutation-based technique makes each individual recall decision simple - if the mutant clone *moCF* of original code fragment *oCF* injected into mutant code base *mioCB* of code base *oCB* is "killed" (i.e., *(oCF, moCF)* is detected as a clone pair) by the detector, then its recall for that clone is 1, otherwise it is 0. We can denote this decision by:

$$R_T^{(oCF,moCF)} = \begin{cases} 1, & \text{if } \textit{(oCF, moCF)} \text{ is detected by } \textit{T} \text{ in } \textit{mioCB}; \\ 0, & \text{otherwise.} \end{cases}$$

The same mutated code fragment, *moCF* can be randomly injected to the original code base, *oCB* any number of times, producing *n* different mutated / injected versions of *oCB*, say *mioCB1, mioCB2 ... mioCBn*. (The framework randomly injects the same mutant clone several times in order to check sensitivity of tools to clone location). The random fragment selector chooses *m* code fragments (say *oCF1, oCF2 ... oCFm*) from the code base, and each of them will be mutated by each mutation operator *dmOP* producing mutated code fragments *moCF1, moCF2 ... moCFm*. Thus, the recall for mutation operator *dmOP* for tool *T* is given by:

$$R_T^{dmOP} = \frac{\sum_{i=1}^{n*m} R_T^{(oCFi,moCFi)}}{m*n}$$

Similarly, for Type 1 clones, three mutation operators (*mCW, mCC, mCF*) and their combinations *((mCW+mCC+mCF), (mCW+mCC), (mCW+ mCF), (mCC+ mCF))* can be applied to the *m* code fragments (although note that if we allow operator repetition we could theoretically generate any number of combinations), each of which will be injected *n* times to the code base. Thus, recall of tool *T* for Type 1 clones can be defined as:

$$R_T^{Typ1} = \frac{\sum_{i=1}^{n*m*(3+4)} R_T^{(oCFi,moCFi)}}{m*n*(3+4)}$$

The overall recall for tool *T* is the summary of recall for the *l* clone mutation operators and *c* combinations applied *n* times to *m* selected code fragments, given by:

$$R_T^{overall} = \frac{\sum_{i=1}^{n*m*(l+c)} R_T^{(oCFi,moCFi)}}{m*n*(l+c)}$$

For tools that give textual results as clone classes rather than clone pairs, we simply check whether the original and mutated / injected code fragments are in the same class.

5.4 Measurement of Precision

In information retrieval, precision measures the noise in the results, such as irrelevant items appearing in the results of a query [31]. The relationship between recall and precision is an indication of how well a matching engine performs. Ideally, precision should remain high as recall increases, but in practice this is difficult to achieve. The more the constraints of a query are relaxed to retrieve more relevant items (increasing recall), the more noise is present in the results (decreasing precision). Similarly, in clone detection the tunable parameters of a tool might be relaxed to detect a reference clone pair, but this can decrease the precision of the results, both for the reference pair in particular and for the tool in general.

Our precision definition is the usual one of IR research, that is, the number of correctly detected items divided by the total number of items detected. Applying precision measurement in the context of mutation-based analysis, we consider the total set of items to be all clone pairs (if any) reported for the injected mutant clone (the mutation-related clone pairs). We then automatically evaluate how many of the mutation-related clone pairs are valid clone pairs (Section 5.6) and use this as the number of correct items.

Using the notation of the previous subsection, let us say that for a mutated code fragment *moCF* created by mutation operator *dmOP*, a tool *T* reports *k* clone pairs, *(moCF, CF1), (moCF, CF2) ... (moCF, CFk)* in mutant code base *mioCB*. If automatic validation reports that *v* of these are valid, then the unit precision of the tool *T* for the single injection of *moCF* for clone type/mutation operator *dmOP* is as follows:

$$P_T^{dmOP\ w.r.t.\ single\ injection\ of\ moCF} = \frac{v}{k}$$

As before, *moCF* may be injected *n* different times, and there may be *m* different code fragments selected for mutation using mutation operator *dmOP*. Thus, the precision of tool *T* for mutation operator *dmOP* is given by:

$$P_T^{dmOP} = \frac{\sum_{i=1}^{n*m} v_i}{\sum_{i=1}^{n*m} k_i}$$

For Type 1 clones, the 3 mutation operators and 4 combinations may be applied *n* times to the *m* code fragments. Thus, precision for Type 1 clones can be calculated as:

$$P_T^{Typ1} = \frac{\sum_{i=1}^{n*m*(3+4)} v_i}{\sum_{i=1}^{n*m*(3+4)} k_i}$$

For overall precision of tool *T* over *l* mutation operators and *c* combinations applied *n* times to *m* code fragments, we have:

$$P_T^{overall} = \frac{\sum_{i=1}^{n*m*(l+c)} v_i}{\sum_{i=1}^{n*m*(l+c)} k_i}$$

Again, for tools that return clone classes rather than clone pairs, we consider all the reported classes associated with the mutated / injected code fragment and form clone pairs from the member fragments of the clone classes.

5.5 Mapping of Code Fragments

In order to measure recall (c.f., Section 5.3), we need to be able to accurately tell if a tool has actually paired a mutant clone with its original, and in order to measure precision (c.f., Section 5.4), we additionally need to know which other reported pairs include the mutant clone. If all tools reported with exactly the same granularity and exactly accurate source locations, then this would be easy - but that is not the case. Thus the framework must be able to determine if a known code fragment is matched or subsumed by fragments in the reported clone pairs.

Definition 4: Fragment containment. We say that a code fragment *CF1* is *contained* by another fragment *CF2* if both are in the same file, and and the range of line numbers of *CF1* is within the range of line numbers of *CF2*. In algorithmic form,

```
boolean isContained(CF CF1, CF CF2) {
   return ((CF1.FileName == CF2.FileName)
       AND (CF1.BeginLine >= CF2.BeginLine)
       AND (CF1.EndLine <= CF2.EndLine))
}
```

Unlike Bellon et al. [10] and Baker [4], we do not consider partial overlapping of clone pairs in this study, because our mutant clones are exactly like the original except for the intended editing mutation. Thus in a partial match, either the non-matching part of the clone pair is the mutated part, in which case the subject tool was unable to match the mutated code lines, or the original part, in which case it failed an exact match. However, if a detected clone pair subsumes the mutant-original clone pair by containment, then we consider that the mutant clone has been detected because similar surrounding code of the original and injected context may also match and be included by the tool. This strict non-overlap binary definition of detection is consistent with the mutation testing tradition of "killed" mutants.

The following algorithm implements our definition of detection for a mutant pair (MP) consisting of clone mutant *moCF* of original fragment *oCF* and a tool *T*'s clone candidate set (CSet) of detected clone pairs *C*:

```
boolean isDetected(MP (oCF, moCF), CSet C) {
   for each clone pair (CF1, CF2) in C {
       if ((isContained(oCF, CF1)
              AND isContained(moCF, CF2))
            OR (isContained(moCF, CF1)
                 AND isContained(oCF, CF2)))
          return True;
   }
   return False;
}
```

To measure precision, we need to find all pairs in *C* for which one of the fragments is the mutant clone *moCF*:

```
CSet validateUs(CF moCF, CSet C){
   CSet ValidateMe = {};
   for each clone pair (CF1, CF2) in C {
       if (isContained(moCF, CF1)
              OR isContained(moCF, CF2))
           ValidateMe = ValidateMe + (CF1, CF2);
   }
   return ValidateMe;
}
```

If the function call *validateUs(moCF, C)* returns the empty set for a run of a particular tool, we do not need to validate any clone pairs. For all non-empty results from *validateUs(moCF, C)* we may need to validate all clone pairs it returns (Section 5.6) except the mutant clone pair itself.

5.6 Validation of Clone Pairs

While recall measurement is completely automatic and no validation of clone pairs is required, to accurately measure precision we need to validate those few clone pairs that are associated with the mutant code fragment. To partially automate this, we have developed a clone pair validator based on our NICAD [37, 41] tool, using standard pretty-printing (to remove Type 1 formatting differences), flexible code normalization (to equalize Type 2 variation in code fragments) and a dissimilarity threshold (to relax the comparison to allow for Type 3 line insertions/deletions) to validate clone pairs. However, unlike NICAD, the validator is not itself a clone detector, both because it works on a specific given clone pair and because it is aware of the mutation operator that was used to produce the mutant clone. Using this information, it can tailor its normalization to make the validation task relatively straightforward.

For example, when only mutation operators of Type 1 clones are used, the validator only applies standard pretty-printing and compares the pair text line-wise. Similarly, when only mutation operators of Type 3 are applied, the validator allows a corresponding dissimilarity threshold [37] in addition to standard pretty-printing of the fragments to relax the similarity between the code fragments of the subject clone pair, and so on. Thus because the validator is aware of the possible differences between the two code fragments, it can accurately measure their real similarity. However, for mutation operators of Type 4 clones, it is not always possible to automatically validate a clone pair. Fortunately, in these few cases the validator can accurately identify those clone pairs for which manual validation is required based on their similarity values.

As part of our first experiment (Section 6), we hand validated more than 500 clone pairs (of all types) as a secondary check. The clone validator neither falsely validated

nor falsely rejected any clone pairs, and correctly tagged the small number of Type 4 pairs requiring manual validation.

5.7 Other Issues

Measurement of Time and Memory Requirements: Measuring time and space requirements of tools seems easy since one needs only to run the tools on large systems, and most tool authors have extensively evaluated these properties of thier tools. Moreover, in our experience time is not a factor, as long as it is reasonable (not in months for example) and the tool gives good quality output (precision and recall). Similarly, memory requirements are generally not a problem since most state-of-the-art tools use linear space. However, our framework can report fine-grained time and memory requirements of subject tools for different clone types, something others have not reported.

Scalability of the Framework: The framework can work with subject systems of any size, depending only on the scalability of the subject tools. While a mutation-based technique inherently implies a large number of tool runs, the framework is built to take advantage of multi-processor machines to balance the load of runs over as many processors as are available. In future we plan to extend this to networks of distributed processors as well.

Adapting Tools to the Framework: The only requirement for adapting a third party tool to the framework is that the tool should be run from the command line (most tools usually do or at least have such an option in addition to their graphical output) and that it should provide a textual report of the detected clones, either in XML format or the usual column-oriented textual format of full file name and begin/end line numbers of the code fragments of the candidate clone pairs. The framework can run subject tools with varying tunable parameters suitable for identifying different types of clones. The framework is language-specific, and currently supports only C, Java and C#. However, since the majority of the framework is language-independent, it is not difficult to add new languages.

6. An Example Use of the Framework

In order to test the framework, both in evaluating a single tool and in comparing a set of tools with different subject systems in a variety of languages, we have conducted several studies of our own NICAD [37] clone detector and its variants in detecting function-level clones. In the following we briefly report our findings.

Subject Code Bases: To test the framework with different code bases and languages, we have chosen several open source subject systems written in C and Java. Table 2 provides a statistical overview of these subject systems (only C and Java files are counted in the calculations).

Single Tool Evaluation: One of the main objectives of the framework is to evaluate the performance of single tools on different types of clones. To examine whether the framework works well with single tools we have used our Basic NICAD variant (see below) and obtained recall and precision values for different types of clones as shown in the *Basic NICAD* column of Table 3.

Table 2. Overview of the Subject Code Bases

Language	Code Base	LOC	Methods
C	Gzip-1.2.4 [20]	8K	117
	Apache-httpd-2.2.8 [1]	275K	4301
	Weltab [9]	11K	123
Java	Netbeans-Javadoc [9]	14K	972
	Eclipse-jdtcore [9]	148K	7383
	JHotDraw 5.4b1 [21]	40K	2399

Table 3. Recall and Precision of the Tools (%)

Type	Mutator	Basic NICAD		FlexP NICAD		Full NICAD	
		Rec.	Prec.	Rec.	Prec.	Rec.	Prec.
Type 1	mCW	100	100	100	100	100	100
	mCC	100	100	100	100	100	100
	mCF	100	100	100	100	100	100
Type 1 overall		100	100	100	100	100	100
Type 2	mSRI	29	96	32	92	100	100
	mARI	31	95	28	91	100	95
	mRPE	28	94	31	95	100	97
Type 2 overall		29	94	27	94	100	97
Type 3	mSIL	96	97	95	97	100	98
	mSDL	96	97	95	97	100	98
	mILs	96	84	91	76	100	95
	mDLs	96	82	93	78	100	94
	mMLs	94	84	92	75	100	96
Type 3 overall		95	85	94	81	100	96
Type 4	mRDS	71	82	66	82	76	90
	mROS	69	83	67	79	78	89
Type 4 overall		67	81	67	79	77	89
Total Overall		87	90	84	89	96	95

Multiple Tool Comparison: Next we wanted to see how the framework would handle multiple tools at a time on the same set of mutants. For this test we used three different versions of NICAD. NICAD has three variants. The first is *Basic NICAD*, which is largely language-independent, applies standard pretty-printing to the source and uses a sequence matching algorithm with dissimilarity thresholds for detecting clones. (This variant was used in a recent large empirical study [41] with good results.) The second variant, Flexible Pretty-Printed NICAD or *FlexP NICAD* [37], adds "flexible" pretty-printing, which reformats to isolate potential changes to different lines, and then uses the same sequence matching algorithm with dissimilarity thresholds. The third variant, *Full NICAD*, utilizes the entire range of NICAD capabilities, including flexible pretty-printing, flexible code normalization, clone filtering and the sequence matching algorithm with dissimilarity thresholds. Our objective was to evaluate and compare the relative capabilities of these three variants using the proposed framework.

Although we have run the experiment with several subject systems, we only have room for a sample of the results.

Table 3 shows the results that we obtained using the framework using *Weltab* [9] as the subject system with each of the three NICAD variants as subject tools. In this experiment, 20 functions were randomly selected from *Weltab* and each of the mutation operators was applied 500 times on those 20 functions, yielding 10,000 mutant code bases. Combinations of the mutation operators were also used. These results are typical of those obtained for NICAD and its variants on other systems. In these results we can see that *Basic NICAD* has relatively poorer recall for clones generated by mutation operators of Type 2 clones (which is not surprising since it was not designed to detect them). *FlexP NICAD* is also not a good choice for these, and thus full *NICAD* is the best option, although there is a cost to its language-specific transformation rules.

7. Related Work

To our knowledge ours is the first attempt to provide a fully automated means for evaluating and comparing clone detection tools. However, several tool authors have done partial evaluation of the precision of their tools, and a number of experiments have attempted to compare tools in terms of precision, recall, time and space. In this section, we outline their evaluation strategies and compare them with ours.

Tool Comparison Experiments: Bailey and Burd [13] compared three state-of-the-art clone detection and two plagiarism detection tools using hand validation of clone candidates. Although they were able to verify all of the clone candidates, the modest size of the subject system, focus on preventive maintenance tasks and reliance on unguided subjective judgment limits the generality of their conclusions.

In an attempt to overcome the limitations of Burd and Bailey's study, Bellon et al. [9, 10] conducted a larger tool comparison experiment on the three clone detection tools used in the Burd and Bailey study as well as three additional tools, and with a much larger and more diverse set of subject systems in C and Java and totalling 850 KLOC. While their study is the most extensive to date, only a small proportion of the clone candidates (about 2%) were oracled, and a number of other factors may have influenced the results [4]. Bellon's framework has since been reused by Koschke et al. [18, 32] and Ducasse et al. [15], with similar results.

Rysselberghe and Demeyer [42] evaluated and compared prototypes of three representative clone detection techniques with respect to portability, kinds of duplication reported, scalability, number of false matches, and number of useless matches. However, their evaluation is qualitative rather than quantitative, focussing on refactoring opportunities rather than general clones. It is also based on small systems and is limited by subjectivity in validation.

Finally, a recent study by Bruntink et al. [12], evaluates several clone detection techniques in identifying known cross-cutting concerns in C programs with homogeneous implementations.

Our work differs from all these experiments in its full automation, its formal clone definition based on an editing taxonomy, its use of mutation and injection to provide large numbers of known and pre-validated clones, its quantitative statistical evaluation based on accepted mutation analysis techniques, and its independence from the target task.

Single Tool Evaluation Strategies: Most text-based techniques [16, 23, 35] have been published with only example-based evaluation, reporting neither precision nor recall. Our own recent text-based tool, NICAD [37] was initially evaluated for precision and recall by manually analyzing and injecting different types of clones into two small C systems. While this evaluation did yield defensible results, the small size of the subject systems and heavy use of manual work does not allow for generalization.

Among the token-based tools [5, 24, 34], only *CP-Miner* [34] was evaluated for precision, by manually examining 100 randomly selected copy/paste segments. Recall was only qualitatively estimated, on the basis of finding more clone pairs than *CCFinder* [24].

The abstract syntax tree (AST) based tool *cpdetector* [18, 32] has been evaluated in terms of both precision and recall for clone Types 1 and 2 by adapting Bellon's framework (and thus inheriting the same limitations). Of the other AST-based tools [8, 17, 22], only *Deckard* [22] was evaluated for precision, by manually validating 100 randomly selected clone groups. While recall was not measured, it reports of detecting more cloned lines than *CloneDr* [8] and *CP-Miner* [34].

Among the metrics-based techniques [30, 31, 36], only Kontogiannis [31] has evaluated the effectiveness of different metrics in his IR-based approach in terms of precision and recall, using subject systems that were manually tagged with known clones by the developers.

The new graph-based approach for semantic clones of Gabel et al. [19] is aimed at the scalability problems of previous graph-based techniques. To estimate precision they randomly sampled 30 clone groups per experiment and manually verified each of them as clones. While they report detecting more clones than *Deckard*, they did not study recall. Previous graph-based methods [29, 33] similarly used random validation for precision but did not report recall.

Our proposed framework provides both recall and precision in an automated, statistically sound way for a range of different types of fine-grained clones. It allows for complete experiments free of subjective opinions and hand analysis, yielding comparable evaluations that overcome the limitations of previous attempts at both validating individual tools and comparing sets of tools.

8. Conclusion

Existing methods for evaluating (and comparing) clone detection tools suffer from several limitations. We have presented an evaluation framework that uses code fragment mutation to artificially create and inject known code clones that can be used to accurately measure recall and precision of clone detection tools. The framework is based on an editing taxonomy that is used to synthesize artificial clones by mimicking developers' typical editing activities in clone creation. The framework is capable of evaluating and comparing recall of clone detection tools for various languages and clone types with no need for manual intervention, and can evaluate precision either automatically using language-dependent validation rules, or semi-automatically using an interface with minimal manual intervention. An example study of evaluating basic NICAD (for single tool evaluation) and three variants of NICAD (for tool comparison) has been successfully conducted using the framework.

As future work, we are already in the planning stages of a large scale tool comparison experiment based on the framework by inviting participation from all current tool authors.

Acknowledgments: This work is supported in part by the Natural Sciences and Engineering Research Council of Canada and by an IBM Faculty Award.

References

[1] The Apache-httpd: `http://httpd.apache.org/` (April 2008)

[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] L. Aversano, L. Cerulo, and Massimiliano Di Penta. How Clones are Maintained: An Empirical Study. In *CSMR*, pp. 81-90, 2007.

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

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

[6] M. Balazinska, E. Merlo, M. Dagenais, B. Lague and K. Kontogiannis. Measuring Clone Based Reengineering Opportunities. In *METRICS*, pp. 292-303, 1999.

[7] M. Balint, T. Girba, Radu Marinescu. How Developers Copy. In *ICPC*, pp. 56-68, 2006.

[8] I. Baxter, A. Yahin, L. Moura and M. Anna. Clone Detection Using Abstract Syntax Trees. In *ICSM*, pp. 368-377, 1998.

[9] S. Bellon and R. Koschke. Detection of Software Clone: Tool Comparison Experiment: `http://www.bauhaus-stuttgart.de/clones/` (December 2007).

[10] 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.

[11] J. Bradbury, J.R. Cordy and J. Dingel. Comparative Assessment of Testing and Model Checking Using Program Mutation. In *Mutation* pp. 210-219, 2007.

[12] 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.

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

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

[15] S. Ducasse, O. Nierstrasz and M. Rieger. On the Effectiveness of Clone Detection by String Matching. *JSME: Research and Practice*, 18(1): 37-58, 2006.

[16] S. Ducasse, M. Rieger and S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *ICSM*, pp. 109-118, 1999.

[17] W. Evans and C. Fraser. Clone Detection via Structural Abstraction. In *WCRE*, pp. 150-159, 2007.

[18] R. Falke, R. Koschke and P. Frenzel. Empirical Evaluation of Clone Detection Using Syntax Suffix Trees. *Empirical Software Engineering*, 43 pp., 2008 (to appear).

[19] M. Gabel, L. Jiang and Z. Su. Scalable Detection of Semantic Clones. In *ICSE*, pp. 321-330, 2008.

[20] The Gzip-1.2.4 `http://www.gzip.org/` (Feb 2008).

[21] The JHotDraw: `http://www.jhotdraw.org/` (June 2006)

[22] 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.

[23] J. Johnson. Substring Matching for Clone Detection and Change Tracking. In *ICSM*, pp. 120-126, 1994.

[24] 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.

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

[26] C. Kapser, and M. Godfrey. Aiding Comprehension of Cloning Through Categorization. In *IWPSE'04*, pp. 85-94, 2004.

[27] M. Kim, L. Bergman, T. Lau and D. Notkin. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *ISESE*, pp. 83–92, 2004.

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

[29] R. Komondoor and S. Horwitz. Using Slicing to Identify Duplication in Source Code. In *SAS*, pp. 40-56, 2001.

[30] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein. Pattern Matching for Clone and Concept Detection. *JASE*, 3(1-2):77-108, 1996.

[31] K. Kontogiannis. Evaluation Experiments on the Detection of Programming Patterns using Metrics. In *WCRE*, pp. 44-54, 1997.

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

[33] J. Krinke. Identifying Similar Code with Program Dependence Graphs. In *WCRE*, pp. 301-309, 2001.

[34] 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.

[35] A. Marcus and J. Maletic. Identification of High-level Concept Clones in Source Code. In *ASE*, pp. 107-114, 2001.

[36] 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.

[37] 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.

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

[39] 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.

[40] C.K. Roy and J.R. Cordy. Towards a Mutation-Based Automatic Framework for Evaluating Clone DetectionTools. In *C3S2E*, Student Poster, pp. 137-140, 2008.

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

[42] F. Rysselberghe and S. Demeyer. Evaluating Clone Detection Techniques from a Refactoring Perspective. In *ASE*, pp. 336-339, 2004.

[43] A. Walenstein, N. Jyoti, J. Li, Y. Yang, A. Lakhotia. Problems Creating Task-relevant Clone Detection Reference Data. In *WCRE*, pp. 285-295, 2003.