Scenario-Based Comparison of Clone Detection Techniques

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

Abstract

Over the last decade many techniques for software clone detection have been proposed. In this paper, we provide a comprehensive survey of the capabilities of currently available clone detection techniques. We begin with an overall survey based on criteria that capture the main features of detection techniques. We then propose a set of hypothetical editing scenarios for different clone types, and evaluate the techniques based on their estimated potential to accurately detect clones that may be created by those scenarios.

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 similar, called code clones. Code clones are often maintained separately and in time may diverge significantly. A difficulty with such duplicated fragments is that if a bug is detected in a code fragment, all fragments similar to it should be checked for the same bug [32]. Duplicated fragments can also significantly increase the work to be done when enhancing or adapting code [35].

Fortunately, several (semi-)automated techniques for detecting code clones have been proposed, and there have been a number of comparison and evaluation studies to relate them. The most recent study, by Bellon et al. [7], 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. Other studies have evaluated clone detection tools in other contexts [25, 8, 40, 41]. 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 [1].

To date no comparative evaluation has considered all of the different techniques available. Each study has chosen a number of state-of-the art tools and compared them using precision, recall, computational complexity and memory use. There is also as yet no third party evaluation of the most recent tools, such as *CP-Miner* [32], *Deckard* [20], *cpdetector* [25], *RTF* [4] and *Asta* [17].

In this paper, we provide an overall comparison and evaluation of all of the currently available clone detection techniques, using both general criteria and a set of edit-based hypothetical scenarios for different clone types. In contrast to previous studies which concentrate on empirically evaluating tools, we aim to identify the essential strengths and weaknesses of both individual techniques in particular and general approaches overall, with a view to providing a complete catalogue of available technology and its potential to recognize "real" clones, that is, those that could be created by the editing of intentionally reused code. Because this paper can fit only a basic summary, not all individual tools can be considered in detail here and our complete study can be found in a technical report [38].

We believe that this is the first study, other than a Dagstuhl presentation report by Koschke [26], that provides a complete comparison of all available clone detection techniques. Our work differs from previous surveys in our use of editing scenarios as a basis for estimating the ability of techniques to detect intentional rather than observed clones, in the evaluation of techniques for which no runnable tools as yet exist, in the inclusion of a number of new methods that have not been previously reviewed, and in the comparison of techniques independent of environment and target language. Our goal is to provide an evaluation indicative of future potential rather than present implementation.

The rest of this paper is organized as follows. After introducing the available clone detection techniques in Section 2, we provide an overall comparison in Section 3 in terms of several general criteria. Section 4 introduces our taxonomy of hypothetical editing scenarios and presents our main result, an analysis of the methods in terms of their estimated ability to detect clones created by each scenario. A discussion on how these survey results can be useful for combining different techniques (or for using a set of tools)

is presented in Section 5. Finally, Section 6 concludes the paper and suggests directions for future work.

2. Overview of Clone Detection Techniques

Many clone detection approaches have been proposed in the literature. Based on the level of analysis applied to the source code, the techniques can roughly be classified into four main categories: textual, lexical, syntactic, and semantic. In this section we briefly introduce and cluster the clone-detection tools and techniques that we later compare by category. Category names from previous taxonomies [25, 7] are mentioned to relate to ours in each case.

2.1 Textual Approaches

Textual approaches (or *text-based techniques*) use little or no transformation/normalization on the source code before the actual comparison, and in most cases raw source code is used directly in the clone detection process.

One of the leading text-based clone detection approaches is that of Ducasse et al. [16], who use string-based Dynamic Pattern Matching (DPM) to textually compare whole lines that have been normalized to ignore whitespace and comments. A similar approach is used by Wettel & Marinescu [46] for finding near-miss clones using scatter plots. Another textual approach is Johnson's [21] redundancy finder, which uses "fingerprints" on substrings of the source code. Manber [33] also uses fingerprints, based on subsequences identified by leading keywords, to identify similar files.

Marcus & Maletic [34] have applied latent semantic indexing (LSI) for finding similar code segments (e.g., high level concept clones such as ADTs) in the source code. This information retrieval approach limits its comparison to comments and identifiers, returning code fragments as clones when there is a high similarity between the identifiers and comments in them. Another text-based approach that claims to be very fast for larger systems is SDD [28].

Some text-based techniques provide tools for visualizing code similarity. *Dotplots* [11] visualizes similar code by tokenizing the code into lines and placing a dot in coordinates (i, j) on a 2D graph if the i^{th} input line matches j^{th} input line. The same approach is used in *Duploc* [16] and *DuDe* [46] for clone visualization.

2.2 Lexical Approaches

Lexical approaches (or *token-based techniques*) begin by transforming the source code into a sequence of lexical "tokens" using compiler-style lexical analysis. The sequence is then scanned for duplicated subsequences of tokens and the corresponding original code is returned as clones. Lexical approaches are generally more robust across minor code changes such as formatting, spacing and renaming than textual techniques.

One of the leading lexical techniques is Kamiya et al.'s *CCFinder* [22]. In *CCFinder*, lines of source files are first divided into tokens by a lexical analyzer, and the tokens of all source files are then concatenated into a single token sequence. Transformation replaces identifiers, constants and other basic tokens with generic tokens representing their language role, and a suffix-tree based sub-string matching algorithm is then used to find common subsequences corresponding to clone pairs and classes. A distributed implementation, *D-CCFinder*, allows the method to scale to very large software systems [31]. *Gemini* [44], based on *CCFinder* visualizes near-miss clones using scatter plots.

A related approach was used earlier in *Dup* [2] for finding exact and parameterized duplicates using suffix-trees. Rather than comparing sequences of individual tokens, *Dup* compares the tokens of individual lines. *RTF* [4] uses a more memory-efficient suffix-array in place of suffix-trees and allows the user to tailor tokenization for better clone detection. Another state-of-the-art token-based technique is *CP-Miner* [32], which uses a frequent subsequence data mining technique to find similar sequences of tokenized statements.

A token- and line-based technique has been used by Cordy et al. [12] to detect near-miss clones in HTML web pages. An island grammar is used to identify and extract all structural fragments of cloning of interest, using pretty-printing to eliminate formatting and isolate differences between clones to as few lines as possible. Extracted fragments are then compared to each other line-by-line using the Unix *diff* algorithm to assess similarity.

2.3 Syntactic Approaches

Syntactic approaches use a parser front-end much like that of a compiler to convert source programs into parse trees or abstract syntax trees (ASTs) which can then be processed using either tree-matching or metrics to find clones.

Tree-based Approaches: Tree-based methods first convert the program to a parse tree or abstract syntax tree (AST) using a parser for the target language. Tree-matching techniques are then used to find similar subtrees, and the corresponding code segments are returned as clone pairs or classes. Variable names, literal values and other leaves (tokens) in the source may be abstracted in the tree representation, allowing for more sophisticated detection of clones.

One of the pioneering AST-based clone detection techniques is Baxter et al.'s *CloneDr* [6]. A compiler generator is used to generate an annotated parse tree (AST). Subtrees are then compared using characterization metrics based on a hash function and tree matching, and corresponding source code fragments are returned as clones. This approach has been adapted into Bauhaus [5] as *ccdiml* (unpublished) using ASTs encoded in IML with similarity metrics, sequence handling and hashing. Yang [47] has proposed a dynamic programming approach for handling syntactic differences in comparing similar subtrees. Wahler et al. [45] find exact

and parameterized clones at a more abstract level by converting the AST to XML and using a data mining technique to find clones. Structural abstraction, which allows for variation in arbitrary subtrees rather than just leaves (tokens), has been used by Evans and Fraser [17] for handling exact and near-miss clones with gaps.

To avoid the complexity of full subtree comparison, recent approaches use alternative tree representations. In the method of Koschke et al. [25], AST subtrees are represented as serialized token sequences (suffix-trees), allowing syntactic clones to be detected more efficiently. A function-level clone detection method based on suffix-trees has been proposed for Microsoft's new Phoenix framework [43].

A novel approach for detecting similar trees has been presented by Jiang et al. [20]. In their method, certain characteristic vectors are computed to approximate the structure of ASTs in a Euclidean space. Locality Sensitive Hashing (LSH) is then used to cluster similar vectors using the Euclidean distance metric and thus find corresponding clones.

Metrics-based Approaches: Metrics-based techniques gather a number of metrics for code fragments and then compare metrics vectors rather than code or ASTs directly. One popular technique involves *fingerprinting functions*, metrics calculated for syntactic units such as a class, function, method or statement that yield values that can be compared to find clones of these syntactic units. In most cases, the source code is first parsed to an AST or CFG (control flow graph) representation to calculate the metrics.

Mayrand et al. [35] use several metrics to identify functions with similar metrics values as code clones. Metrics are calculated from names, layout, expressions and (simple) control flow of functions, and a clone is defined as a pair of whole function bodies with similar metrics values. Patenaude et al. [37] use very similar method-level metrics to extend the Bell Canada Datrix tool to find Java clones.

Kontogiannis [24] uses two ways of detecting clones. One approach uses direct comparison of metrics values as a surrogate for similarity at the granularity of $begin - end$ blocks. A modified version of five well known metrics that capture data and control flow properties is used. The second approach uses a dynamic programming (DP) technique to compare $begin - end$ blocks on a statement-by-statement basis using minimum edit distance. The hypothesis is that pairs with a small edit distance are likely to be clones caused by cut and paste activities. A similar approach is applied by Balazinska et al. [3] in SMC, using a hybrid method that combines characterization metrics with DPM.

Davey et al. [13] detect exact, parameterized and near-miss clones by first computing certain features of code blocks and then training neural networks to find similar blocks based on the features. Metrics-based approaches have also been applied to finding duplicate web pages and clones in web documents [14, 9].

2.4 Semantic Approaches

Semantics-aware methods have also been proposed, using static program analysis to provide more precise information than simply syntactic similarity.

PDG-based Techniques: Program Dependency Graph (PDG)-based approaches [23, 27, 30] go a step further in source code abstraction by considering semantic information encoded in a dependency graph that captures control and data flow information. Given the PDG of a subject program, a subgraph isomorphism algorithm is used to find similar subgraphs which are then returned as clones.

One of the leading PDG-based clone detection methods is proposed by Komondoor and Horwitz [23], which finds isomorphic PDG subgraphs using (backward) program slicing. Krinke [27] uses an iterative approach (k-length patch matching) for detecting maximal similar subgraphs in the PDG. There is also a recent PDG-based tool, GPLAG [30], for plagiarism detection.

Hybrids: In addition to the above, there are also clone detection (and plagiarism) techniques for Lisp-like languages, and Leitao [29] provides a hybrid approach that combines syntactic techniques (using metrics) and semantic techniques (using call graphs) in combination with specialized comparison functions.

3. Comparison of Techniques and Tools

Tables 1 and 2 show an overall summary of the techniques and tools with respect to several general characteristics. The first two columns specify comparison criteria/subcriteria, and the third gives citations for the techniques/tools that match them. In order to provide comparison of both general techniques and individual tools, we gather citations of the same category together using a category annotation, T for text-based, L for lexical (token-based), S for syntactic (tree-based), M for metrics-based and G for graph (PDG)-based, with combinations for hybrids. An asterisk (*) indicates possible limitations in satisfying the criterion.

The first criterion, *Language Paradigm*, indicates the language paradigm targeted by the tool, and the second, *Language Support*, refines this to four particular languages. We can observe that there are very few tools that are aimed at OO-languages (e.g., C++).

The third criterion, *Clone Relation*, addresses how clones are reported– as clone pairs, clone classes, or both. Clone classes can be more useful than clone pairs, for example reducing the number of cases to be investigated by a reengineer when refactoring. Techniques that provide clone classes directly (e.g., *RTF* [4]) may therefore be better for maintenance than those that return only clone pairs (e.g., *Dup* [2]) or require post-processing to group clones into classes (e.g., *CCFinder* [22]).

The fourth criterion, *Clone Granularity*, indicates the granularity of the returned clones – free (i.e., no syntac-

Table 1. Summary of the Surveyed Techniques and Tools

T=text-based, L=lexical/token-based, S=syntactic/tree-based, M=syntactic/metrics-based, G=graph/PDG-based

No.	Criteria	Sub-criteria	Citations and Approaches
1	Language Paradigm	Only Procedural	T [21, 33, 34], L [4], S [47, 19, 25] M [10, 13] G [23, 27, 30]
		Only OOP	S [17], M [37, 3, 36]
		Procedural + OOP	T [16, 28], L [2, 22, 4], S [6, 20, 5]
		Web Applications	L [12], M [14, 9]
		Lisp-like	SMG [29] (hybrid)
2	Language Support	C	T [16, 46, 21, 28, 34, 33], L [22, 2, 4], S [6, 47, 19, 20, 25, 5], M [35, 24, 10] G [23, 27, 30]
		C++	T [16]*, S [5] L [22]
		Java	T [16, 46, 28], L [22, 2, 4], S [6, 17, 20, 5], M [37, 3, 36]
		COBOL	T [16], L [22] S [5]
3	Clone Relation	Directly Clone Pair	T [16, 46, 28], L [2, 22], S [6, 25, 5], M [35, 9, 10], G [23, 27]
		Directly Clone Class	T [21]* (file) T [34]* (ADT), L [4, 12], S [20], M [3], G [30]
		CC in post-processing	T [16], L [22], S [6], M [3], G [23]
4	Clone Granularity	Free	T [16, 21, 46, 28, 34], L [22, 2, 4], S [6, 17, 19, 20, 25, 5], G [27, 23]*
		Fixed	T [33]* (file), L [12] S [43] [47] (file), M [35, 3, 13, 9, 10, 13]
5	Clone Similarity/Types	Exact Match	(almost all)
		Param/Renamed Match	L [2, 22], S [25, 6, 5]
		Near-Miss	T [16, 46, 28, 33], L [22, 4, 32, 12], S [6, 20, 17, 19, 5], M [35, 3, 24, 36, 9, 10], G [23, 27]
		Others	T [34] (ADT), [11] (vis), S [47] (vis)
6	Language Dependency	Lightweight Parsing	T [16, 46, 21, 28, 34, 33], L [12](Island Grm) S [20] (grammar only) M [9]
		Lexer	L [22, 2, 4]
		Lexer and Parser	L [32], S [6, 47, 17, 19, 25, 5], M [35, 3, 10], G [24]
		PDG/CFG Maker	G [23, 27, 30], SMG [29] (call graph)
		Transformation Rules	L [22] (lexical)
7	Text-Processing	Pre-processing	L [12], S [19]
		Post-processing	L [2, 22, 44]
8	Basic Normalization/ Transformation	No changes	T [28, 34] T [46] (whitespace and single brackets) M [9]
		Remove C & Ws	T [16, 21, 46], L [2, 4, 12]
		Ignore C & Ws	L [2] S [6, 20, 47, 17, 19, 5, 25], M [3, 10], G [23, 27, 30]
		Remove C&Ws, Nrm.	T [16]*, L [22, 32]
		Remove C&Ws, Nrm.&Tr.	L [22], M [36]
		Keep C & Ws, Others	T [34] [21]*, M [35]
		Flexible Normalization	L [4], S [20]
9	Code Representation	Filtered Strings	T [16, 46, 28], L [12]
		Filtered Substrings	T [21, 33] (fingerprint)
		Normalized Strings	T [16], L [22, 4] (token sequence)
		Parameterized Strings	L [2] (p-token sequence)
		Word in Context	T [34]
		Metrics/Vectors	S [20] (Char. Vec), M [35] (IRL), M [24, 3, 37, 9, 10]
		Trees	S [6, 47], S [5](IML), S [45, 17] (XML), S [19](string alignment) , S [25, 43](suffix-trees)
		Graph	G [23, 30] (PDG), G [27] (PDG+AST)
		Hybrid	SMG [29] (AST+Metrics+call graph)
10	Comparison Granularity	Line	T [16, 46, 11], L [12] L [2] (p-tokens of line)
		Substring/fingerprint	T [21, 33] (multi-line), [28] (multi-word)
		Identifiers and Comments	T [34]
		Tokens	L [22, 4, 2], S [25, 43] (tokens of suffix trees)
		Statements	L [32], S [45]
		Subtree	S [6, 5, 47, 17, 20, 5]
		Subgraph	G [23, 27, 30]
		Begin-End Blocks	M [24]
		Methods	S [43], M [35, 3, 9, 37, 10]
		Files	T [33], S [47]
11	Comparison Algorithm	Suffix-tree/array/dotplot	T [46], L [2, 22, 4], S [25]
		Data Mining/ IR	T [34] (LSI), L [32] (freq. subseq.), S [45] (freq. itemset)
		Fingerprinting	T [21, 33]
		DMP	T [16], M [35, 3, 10]
		Hash-value Comparison	S [6, 20, 5]
		Graph-Matching	G [27, 30], [23] (slicing)
		Euclidian Distance	M [14]
		Sequence Matching	T [28] (n-neighbor), L [12], S [47, 19] (dynamic prog.)
		Hybrid	SMG [29]

C=comments, Ws=whitespace, Nrm=normalization, Tr=transformation

Table 2. Summary of the Surveyed Techniques and Tools (Continued)

T=text-based, L=lexical/token-based, S=syntactic/tree-based, M=syntactic/metrics-based, G=graph/PDG-based

No.	Criteria	Sub-criteria	Citations and Approaches
12	Computational Complexity	Linear	T [28], L [22, 2, 4], S [25]
		Quadratic (worst case)	T [16], L [12], S [6, 5, 20, 47, 19, 43], M [3, 10, 24](wrt. no. of methods)
		Non-Polynomial	G [27, 23, 30]
13	Heuristics/ Thresholds	On Size of Block	T [16], [28] (4 words), [46], [21] (50 lines), L [2] (15 lines), L [22, 4] (30 tokens)
		On Code Similarity	T [46], L [2, 22, 32, 4, 12], S [6, 20, 5, 17] M [3, 9, 10, 24]
		On Gap Size	T [16, 28, 46], L [32], S [20, 17, 19], M [3]
		On Pruning	T [16, 21]*, [46], L [22, 4]*
14	Output	Only Textual	T [33, 34, 28], S [20, 25], M [10, 24, 24]
		Only Visualization	T [11], L [12, 44], S [47, 17, 19] M [9]
		Both Above	T [16, 46, 21], L [22, 2]
15	Plug-in Support	Yes	T [28, 15, 42] (Eclipse), S [43] (MS Phoenix framework)
16	Empirical Validation	Validated Well	L [22], S [25]
		Moderate Validation	T [16, 46, 34], S [17, 20], M [3, 24], G [30]
		Partially Validated	T [28, 21] L [2, 4, 12], M [9] G [23, 27]
17	Availability of Empirical Results	Yes	S [25]
		Moderate	T [16], L [32, 22], S [20, 17] , M [3, 10]
		Partial/Poor	T [28, 21, 46, 34, 33, 16] L [2, 4, 12], M [9] G [23, 27]
18	Frequently Used Systems	JDK (Java, 204K LOC)	T [28], L [22], S [1, 20], M [3] [7]
		Linux Kernel (C)	L [4] [22], S [20] M [10]
		SNNS 4.1 (C, 115K LOC)	S [25, 1], [7]
		postgsql (C, 235K LOC)	S [1, 25], [7]

Note: The data in these tables is based on published materials, not necessarily the current/future status of the tools.

tic boundaries), fixed (i.e., predefined syntactic boundaries such as method or block) or both. Both granularities have advantages and disadvantages. For example, techniques that return only clones of methods are good for architectural refactoring, but may miss opportunities to introduce new methods for common statement sequences. A tool that handles multiple granularities may be more useful for general reengineering.

The fifth criterion, *Clone Similarity/Types*, considers the kinds of clones a technique can handle. While all techniques can detect exact clones, only *Dup* [2] can directly find parameterized clones. This issue is discussed in detail in the context of edit-based scenarios later in the paper.

The sixth criterion, *Language Dependency*, indicates the kind of language support required for a particular technique. While text-based techniques usually require only lightweight parsing support, other methods can be very language-dependent (e.g., requiring a full parser). We can see that even *CCFinder* [22] requires language-dependent transformation rules. On the other hand, the parse-tree based tool *Deckard* [20] does not require a separate parser, but only a context-free grammar for the target language.

The seventh criterion, *Text-Processing*, refers to any interesting pre- or post-processing (or pretty printing) required other than the usual filtering. For example, we see that Cordy et al. [12] use an island grammar to build a set of all potential clone candidates before comparison.

Noise (e.g., comments) filtering, normalization and transformation of program elements is an important step in clone detection tools, helping both in removing uninteresting clones (filtering), and in finding near-miss clones (normalization and transformation). Our eighth criterion, *Basic Transformation/Normalization*, deals with this issue. Most techniques (except Marcus [34] and *Covet* [35]) either remove comments and whitespace with lightweight parsing or ignore them while generating an AST or flow graphs. We can also see that while text-based techniques such as *Duploc* [16] apply little or no normalization and transformation, others such as *CCFinder* [22] do a lot. Some techniques even provide the user with the option of choosing different normalizations– for example, *RTF* [4] allows several options for tokenizing. We can also note that although source transformation may help in finding near-miss clones, there are really only two methods that apply it, *CCFinder* [22] and Nasehi et al. [36].

The *Code Representation* criterion refers to the internal code representation after filtering, normalization and transformation. The complexity of the detector implementation, the bulk of which is the normalization, transformation and comparison, depends a great deal on the code representation. One should note that we have already generally classified the techniques based on overall level of analysis in Section 2. Here we attempt a finer-grained classification based on the actual representation used in the comparison phase. For example, although a tree-based technique, the actual code representation of *cpdetector* [25] is a serialized token-sequence of AST-nodes, improving the computational and space complexities of the tool from quadratic to linear using a suffix-tree based algorithm.

Different techniques work at different levels of compari-

son granularity, from single source lines to entire AST/PDG subtrees/subgraphs. The tenth criterion, *Comparison Granularity*, refers to the granularity of the technique in the comparison phase. The choice of granularity is crucial to the complexity of the algorithm and the returned clone types, and also determines the kinds of transformation and comparison required. For example, a token-based technique may be more expensive in terms of time and space complexity than a line-based one because a source line generally contains several tokens. On the other hand, a token representation is well suited to normalization and transformation, so minor differences in coding style are effectively removed, yielding more clones. Similarly, although subgraph comparison can be very costly, PDG-based techniques are good at finding more semantics-aware clones.

The choice of algorithm is also a major concern with respect to the time and space complexities, comparison granularity and robustness in detecting near-miss clones. The eleventh criterion, *Comparison Algorithm*, identifies the different algorithms used in clone detection research from other domains. For example, the suffix-tree algorithm computes all of the same subsequences in a sequence composed of a fixed alphabet (e.g., characters, tokens, hash values of lines) in linear time and space, but can only handle exact sequences. On the other hand, data mining algorithms are well suited to handle arbitrary gaps in the subsequences.

The overall computational complexity of a clone detection technique is a major concern, since a practical technique should scale up to detect clones in large software systems with millions of lines of code. The complexity of an approach depends on the kinds of transformations and the comparison algorithm used. The criterion *Computational Complexity* indicates the overall computational complexity of a particular method.

The criterion *Heuristics/Thresholds* indicates whether there are any thresholds and/or heuristics used by a particular method. We see that most of the techniques use similarity-based heuristics with several kinds of thresholds.

The criterion *Output* indicates the kind of output supported by the particular tool. Some tools provide cloning information visually (e.g., *Dotplot* [11]), some provide only textual reports (e.g., [28]), and some provide both (e.g., *Duploc* [16]). The criterion *Plug-in Support* indicates whether there is documented IDE support for the method/tool. Only a few methods provide direct IDE support.

Empirical validation of tools is important, especially in terms of precision, recall and scalability. The criterion *Empirical Validation* hints at the kind of validation that has been reported for each technique, and *Availability of Empirical Results* notes whether the results of the validations are available. The last criterion, *Frequently Used Systems*, notes which common systems have been used in validation.

The last three criteria can assist in choosing a well validated tool/technique, in comparing a new tool with one that has existing empirical results, or in choosing a commonly used subject system as a benchmark. They may also encourage empirical studies on promising tools and techniques that are as yet inadequately validated.

4. Scenario-Based Evaluation

Clone detection techniques are often inadequately evaluated, and only a few studies have looked at some of the techniques and tools [7, 40, 41, 8]. Of these, the Bellon et al. [7] study is the most extensive to date, with a quantitative comparison of six state-of-the-art techniques, essentially all of those with tools targeted at the C and Java languages. 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 [1]. 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, we have taken a predictive, scenario-based approach. We have designed a small set of hypothetical program editing scenarios representative of typical changes to copy/pasted code (Figure 1). Each scenario induces a clone type in the sense of Koschke [26], with additional distinguishing refinements. We assume that our primary intention is to find true clones, that is, those that actually result from copy-and-edit reuse of code.

From a program comprehension point of view, finding such true clones is useful since understanding a representative copy from a clone group assists in understanding all copies in that group [21]. Moreover, replacing all the detected similar copies of a clone group by a function call to the representative copy (i.e., refactoring) can potentially improve understandability, maintainability and extensibility, and reduce the complexity of the system [18].

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 assumed the most lenient settings of any tunable parameters of the techniques. Thus, this is not an actual evaluation, rather it provides an overall picture of the potential of each technique in handling clones resulting from each of the scenarios. Our comparison is not intended to be a concrete experiment, and could not be comprehensive or truly predictive if it were cast as one, bound to target languages, platforms and implementations.

Table 3 provides an overall summary of the results of our evaluations, where the symbols represent an estimate of the ability of each method to accurately detect each (sub-)

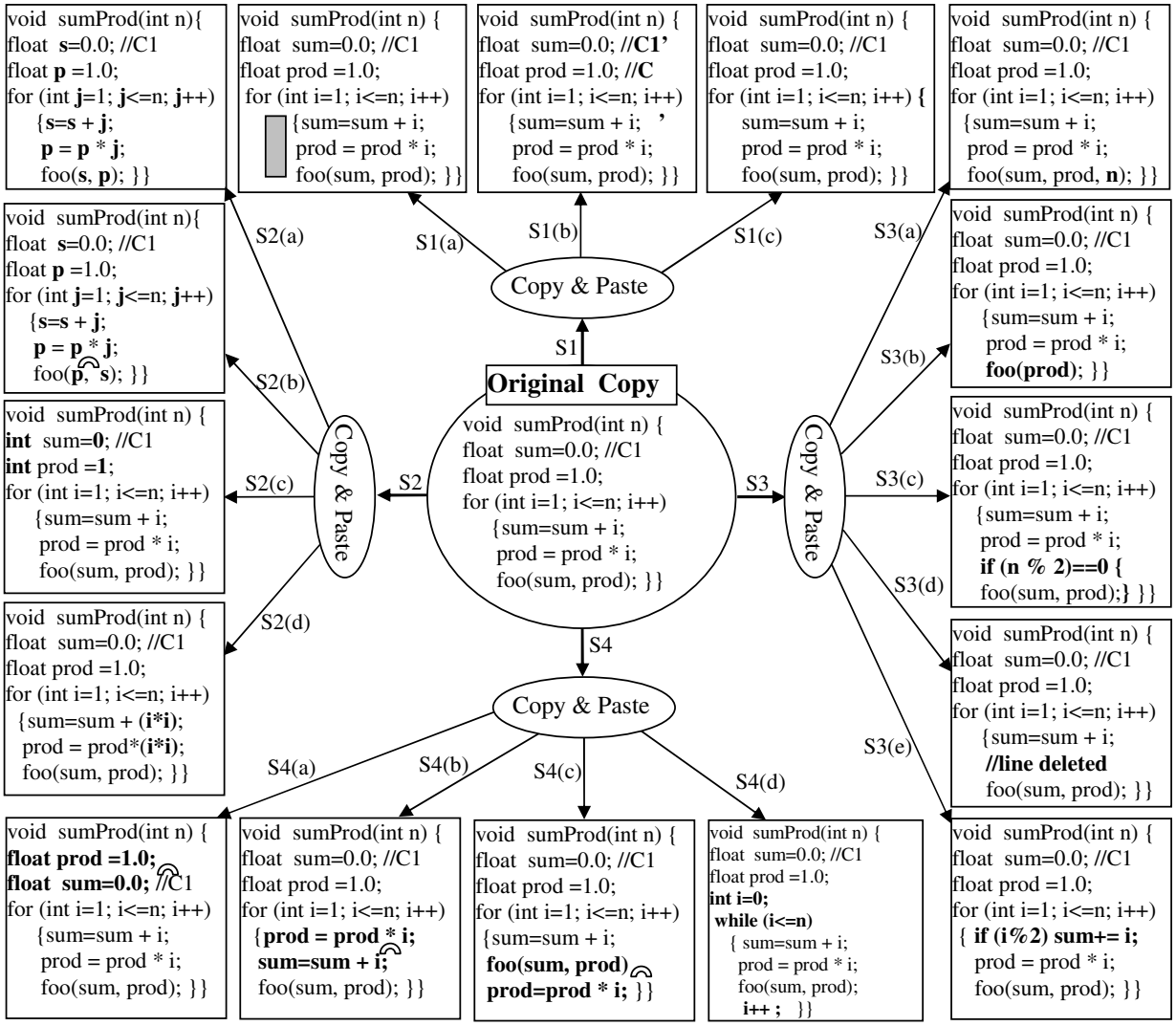

Figure 1. Taxonomy of Editing Scenarios for Different Clone Types

scenario with both high precision and high recall. The summary column on the right estimates the scenario coverage of the technique as a percentage of scenarios potentially detected, counting $\ominus$ (low) and above as potential detection. An asterisk (*) indicates a method with special limitations such as whole file comparison, visualization only, or plagiarism detection only. In the following paragraphs, we consider each scenario and outline our reasoning in estimating the ability of the techniques to accurately detect them.

Scenario 1: *A programmer copies a function that calculates the sum and product of a sequence of numbers* $(1...n)$ *three times, making changes in whitespace in the first fragment (S1(a)), changes in commenting in the second (S1(b)), and changes in formatting in the third (S1(c)) (Figure 1).*

An ideal clone detection technique should recognize all three copy/pasted/modified fragments as clone pairs with the original or form a clone class for them. The third column of Table 3 summarizes how well each technique is likely to work in these scenarios. We expect that only text-based *Duploc* and *DuDe* and token-based *Dup* will find scenarios S1(a) and S2(b) with high precision as they compare the filtered text (e.g., whitespace and comments are removed) line-by-line. However, these line-based techniques are sensitive to format alternations and thus, may not detect scenario S1(c). On the other hand, Marcus's text-based LSI approach and metrics-based *Covet* may not work well for scenario S1(b) as these techniques compare comments either directly (Marcus) or in metrics (*Covet*).

Tree-based techniques (e.g., *cpdetector*) ignore formatting differences and comments and should detect these scenarios very well if they look for exact subtrees without ignoring tree-leaves (in most cases they ignore leaves). However, some tree-based techniques use alternative representations of the parse-tree/AST (e.g., *Deckard* works on charac-

Table 3. Scenario-Based Evaluation of the Surveyed Clone Detection Techniques

● *very well* ◓ *well* ◒ *medium* ⊖ *low* ⊙ *probably can* ○ *probably can't* ∘ *can't*

Approach	Citation	Scenario 1			Scenario 2				Scenario 3					Scenario 4				Scen. 5		Coverage
		a	b	c	a	b	c	d	a	b	c	d	e	a	b	c	d	a	b	(percent)
Text-based	Johnson [21]*	◓	◓	◓	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	16
	Duploc [16]	●	●	⊙	∘	∘	∘	∘	◓	◓	∘	∘	◒	∘	∘	∘	∘	⊙	◒	33
	sif [33]*	◒	◒	◒	∘	∘	⊙	⊙	⊙	⊙	∘	∘	∘	⊙	⊙	⊙	∘	∘	∘	16
	DuDe [46]	●	●	⊙	∘	∘	∘	∘	◓	◓	◒	◒	◒	∘	∘	∘	∘	∘	∘	39
	SDD [28]	◓	◓	◒	∘	∘	∘	∘	◓	◓	⊖	⊖	⊖	∘	∘	∘	∘	∘	∘	44
	Marcus [34]*	◒	∘	◒	∘	∘	◒	⊖	◒	◒	∘	∘	∘	◒	◒	◒	∘	∘	∘	50
Lexical/ Token-based	Dup [2]	●	●	∘	●	∘	◒	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	◒	◒	33
	CCFinder [22]	◒	◒	◒	◒	◒	◒	∘	⊙	⊙	∘	∘	∘	∘	∘	∘	∘	∘	∘	33
	Gemini [44]*	◒	◒	◒	◒	◒	◒	∘	◒	◒	◒	◒	◒	∘	∘	∘	∘	∘	∘	61
	RTF [4]	◓	◓	◓	◓	◓	◓	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	◓	◓	44
	CP-Miner [32]	◓	◓	◓	◓	◓	◓	⊖	◓	◓	◒	◒	◒	∘	∘	∘	∘	◓	◓	78
Syntactic/ AST/ Tree-based	CloneDr [6]	◓	◓	◓	◓	◓	◓	⊖	◒	◒	∘	∘	⊖	⊖	∘	∘	∘	◒	⊖	72
	Asta [17]	◓	◓	◓	◒	◒	◒	◒	◒	◒	∘	∘	∘	∘	∘	∘	∘	◒	◒	61
	Yang [47]*	◓	◓	◓	◓	◓	◓	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	33
	cpdetector [25]	●	●	●	◓	◓	◓	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	◓	◓	44
	Deckard [20]	◓	◓	◓	◓	◓	◓	∘	◒	◒	◒	◒	◒	∘	∘	∘	∘	◓	◓	72
	Tairas [43]	◓	◓	◓	○	○	◒	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	◒	∘	22
	CloneDetection[45]	◓	◓	◓	◓	◓	◓	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	⊙	∘	33
Syntactic/ Metrics-based	Konto [24]	◒	◒	◒	◒	◒	◒	⊖	⊖	⊖	⊖	⊖	⊖	◒	◒	◒	⊖	⊖	∘	94
	Covet [35]	◒	∘	◒	◒	◒	◒	◒	◒	◒	⊖	⊖	⊖	◒	◒	⊖	∘	⊖	∘	83
	Davey [13]	◒	◒	◒	◒	◒	◒	◒	◒	◒	⊙	⊙	⊙	◒	◒	⊖	○	∘	∘	67
Semantical/ Graph-based	Duplix [27]	◓	◓	◓	◓	◓	◓	◒	◒	◒	◒	◒	◒	◓	◒	⊖	∘	◒	∘	89
	KomoRag [23]	◓	◓	◓	◓	◒	◒	⊖	⊖	⊙	⊖	⊙	⊙	◓	◓	⊖	∘	◓	∘	72
	GPLAG [30]*	◓	◓	◓	◓	◒	◒	⊖	⊖	⊖	◒	◒	◒	◓	◓	⊖	◒	⊖	∘	94
Potential Hybrids	Best Rater	[16]	[16]	[25]	[2]	[4]	[25]	[17]	[46]	[28]	[46]	[20]	[20]	[27]	[30]	[34]	[30]	[25]	[20]	
	2nd Best Rater	[2]	[2]	[6]	[25]	[25]	[32]	[35]	[17]	[17]	[20]	[46]	[32]	[30]	[23]	[24]	[24]	[20]	[2]	

Note: Evaluation results reflect the estimated abilities of the underlying technique of a tool, not necessarily the tool itself.

teristic vectors of the parse-tree) and may not detect them accurately. Moreover, a recent study [39] shows that an AST-based exact matching function clone detection technique [43] can even miss some exact function clones detected by a text-line based technique [39]. Metrics-based techniques may return the same metrics values for other scenarios of this study and thus return false positives in our sense. Graph-based techniques, on the other hand give lots of variants of ideal clones and thus cannot detect these scenarios accurately in our view.

Scenario 2: *The programmer makes four more copies of the function, using a systematic renaming of identifiers and literals in the first fragment (S2(a)), renaming the identifiers (but not necessarily systematically) in the second fragment (S2(b)), renaming data types and literal values (but not necessarily systematically) in the third fragment (S2(c)), and replacing some parameters with expressions in the fourth fragment (S2(d)) (Figure 1).*

Again, an ideal clone detection technique should detect all four modified fragments as clone pairs with the original function or should form a clone class for them. The fourth column of Table 3 summarizes how well each technique may work on these scenarios. We rate *Dup* to be robust in detecting scenario S2(a) because of its novel use of parameterized suffix-trees. None of the text-based techniques is likely to do well with these scenarios since they normally compare program text without normalization and are therefore fragile to identifier renaming. Token-based techniques can detect scenarios S2(a), S2(b) and S2(c) well, but are likely to also have many false positives due to their identifier normalizations and transformations. Tree-based techniques may also detect scenarios S2(a), S2(b) and S2(c) well as these techniques normally ignore identifiers and literals in comparison. For scenario S2(d), tree-based *Asta* seems to be well suited, as it can apply structural abstraction on arbitrary subtrees. Metrics-and PDG-based techniques can also detect these scenarios, but metrics-based methods may return many false positives because our other scenarios can yield similar metrics values.

Scenario 3: *The programmer makes five more copies of the function and this time makes small insertions within a line in the first fragment (S3(a)), small deletions within a line in the second fragment (S3(b)), inserts some new lines in the third fragment (S3(c)), deletes some lines from the fourth fragment (S3(d)), and makes changes to some whole lines in the fifth fragment (S2(e)) (Figure 1).*

We would hope that an ideal clone detection technique would detect all five fragments as clone pairs with the original or form a clone class of them. The fifth column of Table 3 summarizes how well we expect that each tech-

nique would work in these scenarios. We expect that text-based *DuDe* and token-based *CP-Miner* are likely to work well with these scenarios. Although *DuDe* is text-based, it can combine small duplication segments to form larger ones by allowing gaps with scatter plot visualization. *CP-Miner* uses a frequent subsequence data mining algorithm which allows it to tolerate gaps in cloned segments. The token-based visualization tool *Gemini* can also identify such scenarios using scatter plot visualization.

Scenario 4: *The programmer makes four more copies of the function and this time reorders the declaration statement in the first fragment (S4(a)), reorders data independent statements in the second (S4(b)), reorders data dependent statements in the third (S4(c)), and replaces a control statement with different one in the fourth (S4(d)) (Figure 1).*

Again, we expect that an ideal clone detection technique should be robust enough to detect such modified code fragments as clone pairs with the original or form a clone class for them. The sixth column of Table 3 summarizes how well each technique is likely to work in these scenarios. It appears that only PDG-based techniques are likely to work well with scenarios S4(a) and S4(b). PDG-based techniques use data and control flow information, which remains unchanged across reordering of declarations and data independent statements. Reordering of data dependent statements may change data and control flow however, so they may not do as well with scenario S4(c). However, Marcus's LSI approach may detect this scenario because the identifiers and comments remain similar in both fragments. To detect scenario S4(d), exhaustive source transformation may be necessary. However, an alternative approach is proposed in the plagiarism detection tool *GPLAG* . If this tool is adapted to detect clones, it might handle scenario S4(d) well.

Scenario 5: *This is a separate scenario where the programmer wants to see how well a technique filters spurious clones i.e., finds syntactic clones (S5(a)), and how efficiently a technique can filter out the code of repetitive regions (e.g., a repetitive sequence of 'switch-case' statements) (S5(b)).*

An ideal clone detection technique should be able to filter out spurious clones and clones of repetitive regions. There are few tools that we can expect to satisfy scenarios S5(a) and S5(b). Almost all text-based techniques (except *Duploc*) do not satisfy these scenarios. Since metrics-based techniques work at a fixed granularity, they may not require spurious clone pruning strategies, although pruning for repetitive regions may still be needed. Recent tools, such as token-based *CP-Miner* and *RTF*, and tree-based *cpdetector* and *Deckard*, apply several strategies for pruning spurious clones (satisfying scenario S5(a)) and clones of repetitive regions (satisfying scenario S5(b)).

5. Towards a Combination of Techniques

Our survey and evaluations are not intended for experts in clone detection, rather our intended audience is potential new users and builders of clone detection-based tools and applications. As a demonstration of how this survey can be helpful, we provide an example combination of different techniques/tools designed to handle all of the scenarios used in this paper. Of course, many other combinations can be derived based on user requirements, both in terms of different scenarios and the techniques used. Such a combination might help one to understand how to design a hybrid method to be robust across all types of clones or how to employ a set of different tools to achieve a better result. The last two rows of Table 3 list the best rated and second best rated techniques for each of the scenarios. Tempering with the properties in Section 3 (especially, from Table 1 and Table 2) and the evaluations in Section 4 (especially, from Table 3), we can select a best choice for each scenario.

For scenarios S1(a), S1(b) and S2(a), the token-and line-based *Dup* [2] seems best, being very good for S2(a) and good for S1(a) and S1(b) while ensuring linear time and space complexity. Although *Duploc* [16] was rated best for S1(a) and S1(b), its (worst-case) quadratic time and space requirements make it a less practical choice. For scenarios S1(c), S2(b) and S2(c), we choose *cpdetector* [25] because it finds syntactic clones in linear time and space. In addition, both *Dup* and *cpdetector* use a suffix-tree algorithm, thus a hybrid of the two might be practical.

For scenarios S2(d), S3(a) and S3(b), we choose *Asta* [17] because it gets a good rating for these scenarios and, like *cpdetector*, it is AST-based, making it a promising choice for a hybrid. For scenarios S3(c), S3(d) and S3(e), *Deckard* seems a good choice. Like *cpdetector* and *Asta*, it is also tree-based and promising for hybridization. For scenarios S4(a), S4(b) and S4(d), adapting the plagiarism detection tool *GPLAG* [30] might be a good choice, as it can detect such scenarios well and it seems to be faster than other PDG-based techniques. *cpdetector* and *Deckard* are also very good in terms of scenarios S5(a) and S5(b), covering several pruning strategies.

Scenario S4(c) seems most appropriate for a fault-detection tool rather than clone detection. However, of the reviewed methods, the LSI approach of Marcus and Maletic [34] would be a good option for this case, although it offers little opportunity for hybridization with our other choices. Thus, the obtained combination is {*Dup*, *cpdetector*, *Asta*, *Deckard* and *GPLAG*}. Several other combinations can easily be obtained based on the results provided in this paper.

6. Conclusion

In this paper, we have focused on detection techniques, providing a concise but comprehensive survey and a hypothetical evaluation based on editing scenarios. A more de-

tailed review of the entire range of clone detection research can be found in our technical report [38], and the Dagstuhl report of Koschke [26] provides an excellent brief overview.

We hope that the results of this study may assist new potential users of clone detection techniques in understanding the range of available methods and selecting those most appropriate for their needs. We hope it may also assist in identifying remaining open research questions, avenues for future research, and interesting combinations of techniques.

The evaluation results of this paper are based on estimating the performance of techniques using the most lenient values of all tunable parameters, and thus our findings differ from the results of empirical studies such as Bellon et al. [7]. While in this study our goal was predictive rather than empirical, in future work we plan to undertake a mutation-based controlled experiment using our editing scenarios as a basis for generating thousands of mutants which can be used to empirically compare actual tools on a similar basis.

Acknowledgements: We thank the four anonymous reviewers for their valuable comments and suggestions in improving the paper. We also thank the tool authors who provided useful answers to our queries, and the colleagues who assisted in tuning and clarifying this paper. This work is supported by the Natural Sciences and Engineering Research Council of Canada.

References

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

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

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

[4] H. Basit, S. Pugliesi, W. Smyth, A. Turpin and S.Jarzabek. Efficient Token Based Clone Detection with Flexible Tokenization. In *ESEC/FSE*, pp. 513-515, 2007.

[5] Project Bauhaus. http://www.bauhaus-stuttgart.de.

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

[7] 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.

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

[9] F. Calefato, F. Lanubile and T. Mallardo. Function Clone Detection in Web Applications: A Semiautomated Approach. *J. of Web Eng.*, 3(1):3-21, 2004.

[10] G. Casazza, G. Antoniol, U. Villano, E. Merlo and M. Penta. Identifying Clones in the Linux Kernel. In *SCAM*, pp. 90-97, 2001.

[11] K. Church and J. Helfman. Dotplot: A program for exploring self-similarity in millions of lines for text and code. *J. of American Stat. Ass.*, 2(2):153-174, 1993.

[12] J.R. Cordy, T.R. Dean and N. Synytskyy. Practical Language-Independent Detection of Near-Miss Clones. In *CASCON*, pp. 29-40, 2004.

[13] N. Davey, P. Barson, S. Field and R. Frank. The Development of a Software Clone Detector. *J. App. Soft. Tech.*, 1(3/4):219-236, 1995.

[14] G. Di Lucca, M. Penta and A. Fasolino. An Approach to Identify Duplicated Web Pages. In *COMPSAC*, pp. 481-486, 2002.

[15] Tool Dupman http://sourceforge.net/projects/dupman

[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] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[19] D. Gitchell and N. Tran. Sim: A utility for detecting similarity in computer programs. In *SIGCSE Bulletin*, 31(1): 266-270, 1999.

[20] 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.

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

[22] 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.

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

[24] 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.

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

[26] R. Koschke. Survey of Research on Software Clones. In *Dagstuhl Seminar 06301*, 24pp., 2006.

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

[28] S. Lee and I. Jeong. SDD: High performance Code Clone Detection System for Large Scale Source Code. In *OOPSLA*, pp. 140-141, 2005.

[29] A. Leitão. Detection of Redundant Code Using R^2D^2. *Soft. Qual. J.*, 12(4):361-382, 2004.

[30] C. Liu, C. Chen, J. Han and P. Yu. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis.In *KDD*, pp. 872-881, 2006.

[31] S. Livieri, Y. Higo, M. Matsushita, K. Inoue: Very-Large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder. In *ICSE*, pp. 106-115, 2007.

[32] 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.

[33] U. Manber. Finding Similar Files in a Large File System. In *USENIX Winter*, pp. 1-10, 1994.

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

[35] 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.

[36] S. Nasehi, G. Sotudeh and M. Gomrokchi. Source Code Enhancement Using Reduction of Duplicated Code. In *IASTED SE*, 2007.

[37] J. Patenaude, E. Merlo, M. Dagenais, and B. Lague. Extending Software Quality Assessment Techniques to Java Systems. In *IWPC*, pp. 49-56, 1999.

[38] C.K. Roy and J.R. Cordy. *A Survey on Software Clone Detection Research*. Queen's Technical Report:541, 115pp., 2007.

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

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

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

[42] Tool SimScan. http://www.blue-edge.bg/simscan/

[43] R. Tairas and J. Gray. Phoenix-Based Clone Detection Using Suffix Trees. In *ACM-SE*, pp. 679-684, 2006.

[44] Y. Ueda, T. Kamiya, S. Kusumoto and K. Inoue. On Detection of Gapped Code Clones Using Gap Locations. In *APSEC*, pp. 327-336, 2002.

[45] V. Wahler, D. Seipel, J. Gudenberg and G. Fischer. Clone Detection in Source Code by Frequent Itemset Techniques. In *SCAM*, pp. 128-135, 2004.

[46] R. Wettel and R. Marinescu. Archeology of Code Duplication: Recovering Duplication Chains From Small Duplication Fragments. In *SYNASC*, 8pp., 2005.

[47] W. Yang. Identifying Syntactic Differences Between Two Programs. *Soft.-Prac. and Exper.*, 21(7):739-755, 1991.