Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach

Chanchal K. Roy*,[a], James R. Cordy[a], Rainer Koschke[b]

[a]School of Computing, Queen's University, Canada
[b]University of Bremen, Germany

Abstract

Over the last decade many techniques and tools for software clone detection have been proposed. In this paper, we provide a qualitative comparison and evaluation of the current state-of-the-art in clone detection techniques and tools, and organize the large amount of information into a coherent conceptual framework. We begin with background concepts, a generic clone detection process and an overall taxonomy of current techniques and tools. We then classify, compare and evaluate the techniques and tools in two different dimensions. First, we classify and compare approaches based on a number of facets, each of which has a set of (possibly overlapping) attributes. Second, we qualitatively evaluate the classified techniques and tools with respect to a taxonomy of editing scenarios designed to model the creation of *Type-1*, *Type-2*, *Type-3* and *Type-4* clones. Finally, we provide examples of how one might use the results of this study to choose the most appropriate clone detection tool or technique in the context of a particular set of goals and constraints. The primary contributions of this paper are: (1) a schema for classifying clone detection techniques and tools and a classification of current clone detectors based on this schema, and (2) a taxonomy of editing scenarios that produce different clone types and a qualitative evaluation of current clone detectors based on this taxonomy.

Key words: Software Clone, Clone Detection, Comparison, Scenario-based Evaluation

1. Introduction

Reusing code fragments by copying and pasting with or without minor adaptation is a common activity in software development. As a result software systems often contain sections of code that are very similar, called *code clones*. Previous research shows that a significant fraction (between 7% and 23%) of the code in a typical software system has been cloned [8, 99]. While such cloning is often intentional [64] and can be useful in many ways [3, 61], it can be also be harmful in software maintenance and evolution [56]. For example, if a bug is detected in a code

*Corresponding author.

Email addresses: croy@cs.queensu.ca (Chanchal K. Roy), cordy@cs.queensu.ca (James R. Cordy), koschke@informatik.uni-bremen.de (Rainer Koschke)

fragment, all fragments similar to it should be checked for the same bug [84]. Duplicated fragments can also significantly increase the work to be done when enhancing or adapting code [87]. Many other software engineering tasks, such as program understanding (clones may carry domain knowledge), code quality analysis (fewer clones may mean better quality code), aspect mining (clones may indicate the presence of an aspect), plagiarism detection, copyright infringement investigation, software evolution analysis, code compaction (for example, in mobile devices), virus detection, and bug detection may require the extraction of syntactically or semantically similar code fragments, making clone detection an important and valuable part of software analysis [102].

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. [18], 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 [72, 22, 105, 106]. 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 [4]. 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* [84], *Deckard* [52], *cpdetector* [72], *RTF* [12], *Asta* [42] and NICAD [104].

In this paper, we provide a comprehensive qualitative comparison and evaluation of all of the currently available clone detection techniques and tools in the context of a unified conceptual framework. Beginning with a basic introduction to clone detection background and terminology, we organize the current techniques and tools into a taxonomy based on a generic clone detection process model. We then classify, compare and evaluate the techniques and tools in two different dimensions.

First, we perform a classification and overall comparison with respect to a number of facets, each of which has a set of (possibly overlapping) attributes. Second, we define a taxonomy of editing scenarios designed to create *Type-1*, *Type-2*, *Type-3*, and *Type-4* clones, which we use to qualitatively evaluate the techniques and tools we have previously classified. In particular, we estimate how well the various clone detection techniques may perform based on their *published* properties (either in the corresponding published papers or online documentation). In order to estimate maximal potential, we have assumed the most lenient settings of any tunable parameters of the techniques and tools. Thus, this is not an actual evaluation, rather it provides an overall picture of the potential of each technique and tool 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 and qualitative if it were cast as one, bound to target languages, platforms and implementations. Finally, we provide two examples of how one might use the results of this study to identify one or more appropriate clone detectors given a set of constraints and goals.

In contrast to previous studies, which concentrate on empirically evaluating tools, we aim to identify the essential strengths and weaknesses of both individual tools and techniques and alter-

native approaches in general. Our goal is to provide a complete catalogue of available technology and its potential to recognize "real" clones, that is, those that could be created by the the editing operations typical of actual intentional code reuse.

To the best of our knowledge, this paper is the first study of the area, other than Koschke's recent overview [70, 69, 73] and our own short conference paper [103], that provides a complete comparison of all available clone detection techniques. For an even more complete in-depth overview of the area, readers are referred to our recent technical report [102].

Our work particularly 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 techniques and tools that have not been previously reviewed, and in the comparison of techniques independent of environment and target language. Our goal is not only to provide the current comparative status of the tools and techniques, but also to make an evaluation indicative of future potential (e.g., when one aims to develop a new hybrid technique) rather than simply present implementation.

The rest of this paper is organized as follows. After introducing some background terms in Section 2, we provide a general overview of the clone detection process in Section 3. We present the available clone detection techniques in the form of a taxonomy in Section 4, and based on the taxonomy, Section 5 presents an overall comparison of the techniques and tools in terms several general criteria organized into facets. Section 6 introduces our taxonomy of hypothetical editing scenarios and presents our qualitative evaluation result, an analysis of the techniques and tools in terms of their estimated ability to detect clones created by each scenario. An example discussion on how the results of this study can be useful to a potential user or tool builder is presented in Section 7. Section 8 relates our work to that of others, and finally, Section 9 concludes the paper and suggests directions for future work.

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, or sequence of statements. 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) = f(CF2)* where *f* is the similarity function (see *clone types* below). Two fragments that are similar to each other form a *clone pair* $(CF1, CF2)$, and when many fragments are similar, they form a *clone class* 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 based on 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. In the following we provide the types of clones based on both the textual (Types 1 to 3) [18] and functional (Type 4)[46, 65] similarities:

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.

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

3. Clone Detection Process

A clone detector must try to find pieces of code of high similarity in a system's source text. The main problem is that it is not known beforehand which code fragments may be repeated. Thus the detector really should compare every possible fragment with every other possible fragment. Such a comparison is prohibitively expensive from a computational point of view and thus, several measures are used to reduce the domain of comparison before performing the actual comparisons. Even after identifying potentially cloned fragments, further analysis and tool support may be required to identify the actual clones. In this section, we provide an overall summary of the basic steps in a clone detection process. This generic overall picture allows us to compare and evaluate clone detection tools with respect to their underlying mechanisms for the individual steps and their level of support for these steps.

Figure 1 shows the set of steps that a typical clone detector may follow in general (although not necessarily). The generic process shown is a generalization unifying the steps of existing techniques, and thus not all techniques include all the steps. In the following subsections, we provide a short description of each of the phases.

3.1. Preprocessing

At the beginning of any clone detection approach, the source code is partitioned and the domain of the comparison is determined. There are three main objectives in this phase:

Remove uninteresting parts: All the source code uninteresting to the comparison phase is filtered out in this phase. For example, partitioning is applied to embedded code to separate different languages (e.g., SQL embedded in Java code, or Assembler in C code). This is especially important if the tool is not language independent. Similarly, generated code (e.g., LEX- and YACC-generated code) and sections of source code that are likely to produce many false positives (such as table initialization) can be removed from the source code before proceeding to the next phase [96].

Determine source units: After removing the uninteresting code, the remaining source code is partitioned into a set of disjoint fragments called source units. These units are the largest source fragments that may be involved in direct clone relations with each other. Source units can be at any level of granularity, for example, files, classes, functions/methods, begin-end blocks, statements, or sequences of source lines.

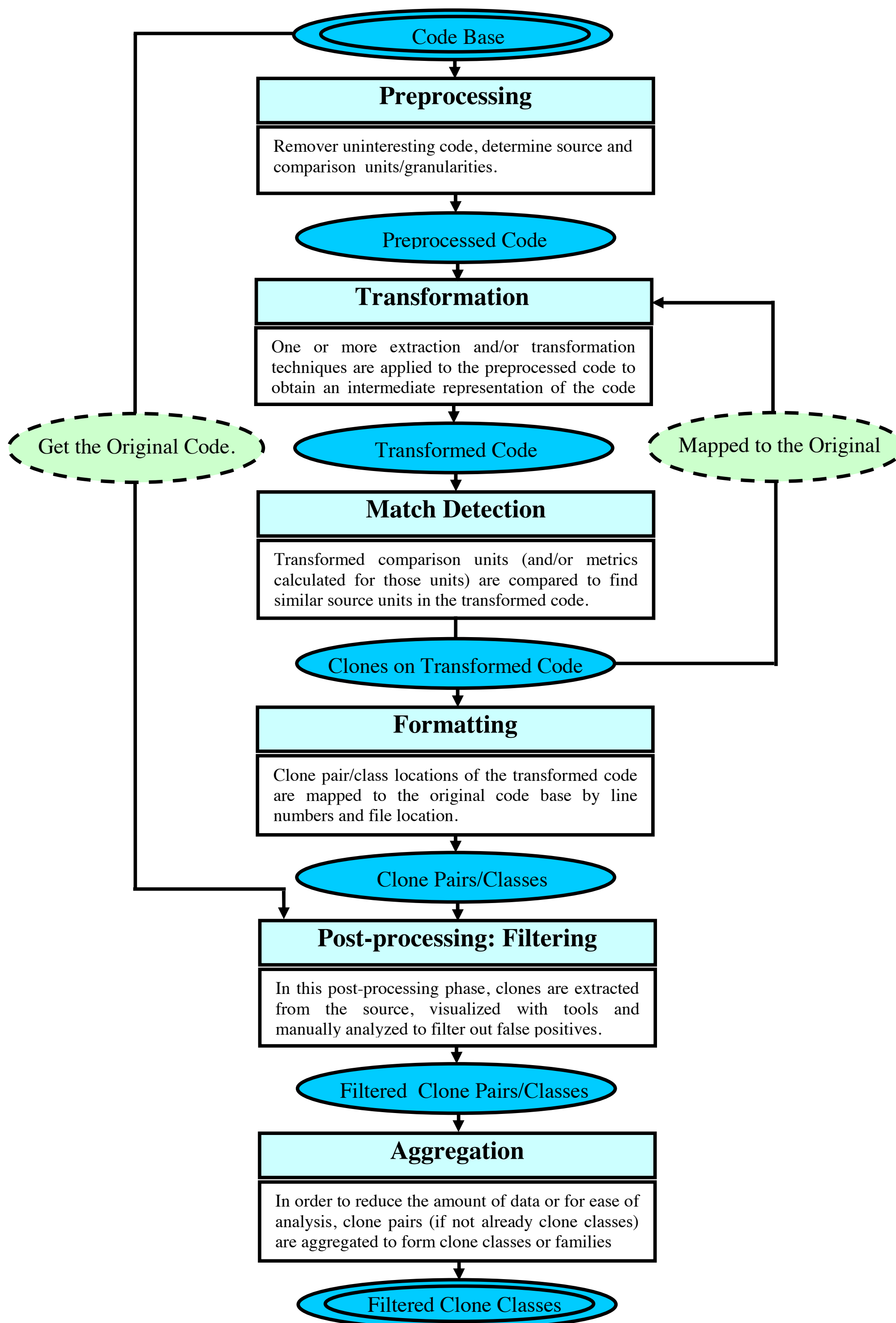

Figure 1: A Generic Clone Detection Process

Determine comparison units / granularity: Source units may need to be further partitioned into smaller units depending on the comparison technique used by the tool. For example, source units may be subdivided into lines or even tokens for comparison. Comparison units can also be derived from the syntactic structure of the source unit. For example, an *if-statement* can be further partitioned into *conditional* expression, *then* and *else* blocks. The order of comparison units within their corresponding source unit may or may not be important, depending on the comparison technique. Source units may themselves be used as comparison units. For example, in a metrics-based tool, metrics values can be computed from source units of any granularity and therefore, subdivision of source units is not required in such approaches.

3.2. Transformation

Once the units of comparison are determined, if the comparison technique is other than textual, the source code of the comparison units is transformed to an appropriate intermediate representation for comparison. This transformation of the source code into an intermediate representation is often called *extraction* in the reverse engineering community.

Some tools support additional normalizing transformations following extraction in order to detect superficially different clones. These normalizations can vary from very simple normalizations, such as removal of whitespace and comments [6], to complex normalizations, involving source code transformations [104]. Such normalizations may be done either before or after extraction of the intermediate representation.

3.2.1. Extraction

Extraction transforms source code to the form suitable as input to the actual comparison algorithm. Depending on the tool, it typically involves one or more of the following steps.

Tokenization: In case of token-based approaches, each line of the source is divided into tokens according to the lexical rules of the programming language of interest. The tokens of lines or files then form the token sequences to be compared. All whitespace (including line breaks and tabs) and comments between tokens are removed from the token sequences. *CCFinder* [59] and *Dup* [6] are the leading tools that use this kind of tokenization on the source code.

Parsing: In case of syntactic approaches, the entire source code base is parsed to build a parse tree or (possibly annotated) abstract syntax tree (AST). The source units to be compared are then represented as subtrees of the parse tree or the AST, and comparison algorithms look for similar subtrees to mark as clones [15, 113, 116]. Metrics-based approaches may also use a parse tree representation to find clones based on metrics for subtrees [66, 87].

Control and Data Flow Analysis: Semantics-aware approaches generate program dependence graphs (PDGs) from the source code. The nodes of a PDG represent the statements and conditions of a program, while edges represent control and data dependencies. Source units to be compared are represented as subgraphs of these PDGs. The techniques then look for isomorphic subgraphs to find clones [65, 75]. Some metrics-based approaches use PDG subgraphs to calculate data and control flow metrics [66, 87].

3.2.2. Normalization

Normalization is an optional step intended to eliminate superficial differences such as differences in whitespace, commenting, formatting or identifier names.

Removal of whitespace: Almost all approaches disregard whitespace, although line-based approaches retain line breaks. Some metrics-based approaches however use formatting and layout as part of their comparison. Davey et al. [31] use the indentation pattern of pretty printed source text as one of the features of their attribute vectors, and Mayrand et al. [87] use layout metrics such as the number of non-blank lines.

Removal of comments: Most approaches remove and ignore comments in the actual comparison. However, Marcus and Maletic [86] explicitly use comments as part of their concept similarity method, and Mayrand et al. [87] use the number of comments as one of their metrics.

Normalizing identifiers: Most approaches apply an identifier normalization before comparison in order to identify parametric *Type-2* clones. In general, all identifiers in the source code are replaced by the same single identifier in such normalizations. However, Baker [6] uses an order-sensitive indexing scheme to normalize for detection of consistently renamed *Type-2* clones.

Pretty-printing of source code: Pretty printing is a simple way of reorganizing the source code to a standard form that removes differences in layout and spacing. Pretty printing is normally used in text-based clone detection approaches to find clones that differ only in spacing and layout. Cordy et al. [28] use an island grammar [91] to generate a separate pretty-printed text file for each potentially cloned source unit.

Structural transformations: Other transformations may be applied that actually change the structure of the code, so that minor variations of the same syntactic form may be treated as similar [59, 92, 104]. For instance, Kamiya et al. [59] remove keywords such as `static` from C declarations.

3.3. Match Detection

The transformed code is then fed into a comparison algorithm where transformed comparison units are compared to each other to find matches. Often adjacent similar comparison units are joined to form larger units. For techniques/tools of fixed granularity (those with a predetermined clone unit, such as a function or block), all the comparison units that belong to the target granularity clone unit are aggregated. For free granularity techniques/tools (those with no predetermined target clone unit) aggregation is continued as long as the similarity of the aggregated sequence of comparison units is above a given threshold, yielding the longest possible similar sequences.

The output of match detection is a list of matches in the transformed code which is represented or aggregated to form a set of candidate clone pairs. Each clone pair is normally represented as the source coordinates of each of the matched fragments in the transformed code.

In addition to simple normalized text comparison, popular matching algorithms used in clone detection include suffix-trees [68, 88, 6, 59], dynamic pattern matching (DPM) [41, 66] and hash-value comparison [15, 87].

3.4. Formatting

In this phase, the clone pair list for the transformed code obtained by the comparison algorithm is converted to a corresponding clone pair list for the original code base. Source coordinates of each clone pair obtained in the comparison phase are mapped to their positions in the original source files.

3.5. *Post-processing / Filtering*

In this phase, clones are ranked or filtered using manual analysis or automated heuristics.

Manual Analysis: After extracting the original source code, clones are subjected to a manual analysis where false positive clones or spurious clones [72] are filtered out by a human expert. Visualization of the cloned source code in a suitable format (e.g., as an HTML web page [104]) can help speed up this manual filtering step.

Automated Heuristics: Often heuristics can be defined based on length, diversity, frequency, or other characteristics of clones in order to rank or filter out clone candidates automatically [59, 58].

3.6. *Aggregation*

While some tools directly identify clone classes, most return only clone pairs as the result. In order to reduce the amount of data, perform subsequent analyses or gather overview statistics, clones may be aggregated into clone classes.

4. Overview of Clone Detection Techniques and Tools

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 summarize the state of the art in automated clone detection by introducing and clustering available clone-detection tools and techniques by category.

The techniques can be distinguished primarily by the type of information their analysis is based on and the kinds of analysis techniques that they use. Table 1 provides a high-level overview of the techniques and tools in the form of a taxonomy where the first column shows the underlying approach of the tools/techniques, the second column shows their one sentence description, the third column either shows the name of the corresponding tool or (if no tool name is found) the last name of the first author has been used as the tool name, and the fourth column shows the corresponding citation(s).

4.1. *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. Johnson pioneered text-based clone detection. His approach [53, 54] uses "fingerprints" on substrings of the source code. First, code fragments of a fixed number of lines (the window) are hashed. A sliding window technique in combination with an incremental hash function is used to identify sequences of lines having the same hash value as clones. To find clones of different lengths, the sliding window technique is applied repeatedly with various lengths. Manber [85] also uses fingerprints, based on subsequences marked by leading keywords, to identify similar files.

One of the newer text-based clone detection approaches is that of Ducasse et al. [41, 96]. The technique is based on dot plots. A dot plot – also known as a scatter plot – is a two-dimensional chart where both axes list source entities. In the case of the approach by Ducasse et al., comparison

entities are the lines of a program. There is a dot at coordinate (x, y) if x and y are equal. Two lines must have the same hash value to be considered equal. Dot plots can be used to visualize clone information, where clones can be identified as diagonals in dot plots. The detection of clones in dot plots can be automated, and Ducasse et al. use string-based dynamic pattern matching on dot plots to compare whole lines that have been normalized to ignore whitespace and comments. Diagonals with gaps indicate possible *Type-3* clones, and a pattern matcher is run over the matrix to find diagonals with holes up to a certain size.

An extension of the Ducasse et al. approach is used by Wettel & Marinescu [114] to find near-miss clones using dot plots. Starting with lines having the same hash value, the algorithm chains together neighboring lines to identify certain kinds of *Type-3* clones. SDD [78] is another similar approach that applies an n-neighbor approach in finding near-miss clones.

NICAD [104, 99] is also text-based, but exploits the benefits of tree-based structural analysis based on lightweight parsing to implement flexible pretty-printing, code normalization, source transformation and code filtering. (Thus NICAD is essentially a hybrid technique.)

Marcus and Maletic [86] apply latent semantic indexing (LSI) to source text in order to find high level concept clones, such as abstract data types (ADTs), in the source code. This information retrieval approach limits its comparison to comments and identifiers, returning two code fragments as potential clones or a cluster of potential clones when there is a high level of similarity between their sets of identifiers and comments.

4.2. 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 over minor code changes such as formatting, spacing, and renaming than textual techniques.

Efficient token-based clone detection was pioneered by Brenda Baker. In Baker's tool *Dup*[8, 6], lines of source files are first divided into tokens by a lexical analyzer. Tokens are split into parameter tokens (identifiers and literals) and non-parameter tokens, with the non-parameter tokens of a line summarized using a hashing *functor*, and the parameter tokens are encoded using a position index for their occurrence in the line. This encoding abstracts away from concrete names and values of parameters, but not from their order, allowing for consistently parameter-substituted *Type-2* clones to be found. All prefixes of the resulting sequence of symbols are then represented by a suffix tree, a tree where suffixes share the same set of edges if they have a common prefix. If two suffixes have a common prefix, obviously the prefix occurs more than once and can be considered a clone.

The technique allows one to detect *Type-1* and *Type-2* clones, and *Type-3* clones can be found by concatenating *Type-1* or *Type-2* clones if they are lexically not farther than a user-defined threshold away from each other. These can be summarized using a dynamic-programming technique [9]. Kamiya et al. later extended this technique in *CCFinder* [59], using additional source normalizations to remove superficial differences such as changes in statement bracketing (e.g., `if(a) b=2;` vs. `if(a) {b=2;}`). *CCFinder* is itself used as the basis of other techniques, such as *Gemini* [112], which visualizes near-miss clones using scatter plots, and *RTF* [12], which uses a

Table 1: Taxonomy of Clone Detection Techniques and Tools

	One Sentence Description	Tool/1st Author	References
Text-Based	Hashing of strings per line, then textual comparison	Johnson	[54, 53, 55]
	Hashing of strings per line, then visual comparison using dotplots	Duploc	[41]
	Finds similar files with approximate fingerprints	sif	[85]
	Composes smaller isolated fragments of duplication with scatter-plot	DuDe	[114]
	Data structure of an inverted index and an index with n-neighbor distance concept	SDD	[78]
	Latent semantic indexing for identifiers and comments	Marcus	[86]
	Syntactic pretty-printing, then textual comparison with thresholds	Basic NICAD	[99]
	Syntactic pretty-printing with flexible code normalization and filtering, then textual comparison with thresholds	Full NICAD	[104]
	Transformation to a middle format of atomic instructions and edit distance algorithm	Nasehi	[92]
	Textual comparison with flexible options (e.g., ignore all identifiers)	Simian	[107]
Token-Based	Suffix trees for tokens per line	Dup	[8, 7, 6]
	Token normalizations, then suffix-tree based search	CCFinder(X)	[59, 58]
	Distributed implementation of *CCFinder* for very large systems	*D-CCFinder*	[83]
	Uses CCFinder's non-gapped clones to find gapped clones in interactive and visual way using a gap-and-clone scatter plot	GeX/Gemini	[112, 58]
	Flexible tokenization and suffix-array comparison	RTF	[12]
	Data mining for frequent token sequences	CP-Miner	[84]
	Real-time token comparison in IDEs with suffix-array	SHINOBI	[115]
	Karp-Rabin string matching algorithm with frequency table of tokens	CPD	[29]
	Normalized token comparison integrated with Visual Studio	Clone Detective	[37]
	Normalized token comparison with suffix-tree	clones	[14, 72]
	clones is adapted to detect clones over multiple versions at a time	iClones	[48]
Tree-Based	Hashing of syntax trees and tree comparison	CloneDr	[15]
	Derivation of syntax patterns and pattern matching	Asta	[42]
	Hashing of syntax trees and tree comparison	cdiff	[116]
	Serialization of syntax trees and suffix-tree detection	cpdetector	[72, 43]
	Metrics for syntax trees and metric vector comparison with hashing	Deckard	[52]
	Suffix-tree comparison of AST-nodes	Tairas	[111]
	XML representation of ASTs with frequent itemsets techniques of data mining	CloneDetection	[113]
	XML representation of ASTs and anti-unification/code abstraction	CloneDigger	[20]
	Token sequence of CodeDOM graphs with levenshtein distance	C2D2	[74]
	Token-sequence of AST-nodes and lossless data compression algorithm	Juillerat	[57]
	Subtree comparison obtained from ANTLR	SimScan	[108]
	Like *cpdetector* but works on the nodes of parse-trees	clast	[14]
	Like *CloneDr* with a different intermediate representation [71] of ASTs	ccdiml	[16, 14]
	AST to FAMIX and then tree matching	Coogle	[109]
Metrics-Based	Clustering feature vector of procedures with neural net	Davey	[31]
	Comparing metrics for functions/begin-end blocks	[66, 87, 93, 67, 30, 89, 90, 1, 2]	
	Comparing metrics for web sites	[23, 38]	
Graph-Based	Approximative search for similar subgraphs in PDGs	Duplix [75], GPLAG [81]	
	Searching similar subgraphs in PDGs with slicing	Komondoor	[65]
	Mapping PDG subgraphs to structured syntax and reuse *Deckard*	Gabel	[46]

more memory-efficient suffix-array in place of suffix trees and allows the user to tailor tokenization for better clone detection.

CP-Miner [84] is another state-of-the-art token-based technique, which uses frequent subsequence data mining to find similar sequences of tokenized statements. A token- and line-based technique has also been used by Cordy et al. [28, 110] to detect near-miss clones in HTML web pages. An island grammar is used to identify and extract all structural fragments of cloning 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.

Because syntax is not taken into account, clones found by token-based techniques may overlap different syntactic units. However, using either preprocessing [28, 47, 104] or post-processing [50], clones corresponding to syntactic blocks can be found if block delimiters are known or lightweight syntactic analysis such as island parsing [91] is added.

4.3. Syntactic Approaches

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

Tree-matching Approaches: Tree-matching approaches (or *tree-based techniques*) find clones by finding similar subtrees. 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 tree-matching clone detection techniques is Baxter et al.'s *CloneDr* [15]. A compiler generator is used to generate a constructor for annotated parse trees. Subtrees are then hashed into buckets. Only within the same bucket, subtrees are compared to each other by a tolerant tree matching. The hashing is optional but reduces the number of necessary tree comparisons drastically.

This approach has been adapted by the AST-based clone detectors of Bauhaus [14] as *ccdiml*. The main differences from *CloneDr* are *ccdiml*'s explicit modeling of sequences, which eases the search for groups of subtrees that together form clones, and its exact matching of trees. Yang [116] has proposed a dynamic programming approach for handling syntactic differences in comparing similar subtrees. (*cdiff* is not a clone detection tool in itself but the underlying technique could be used in clone detection). Wahler et al. [113] 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 proposed by Evans et al. [42] 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 approach of Koschke et al. [72, 43], AST subtrees are serialized as AST node sequences for which a suffix tree is then constructed. This idea allows to find syntactic clones at the speed of token-based techniques. A function-level clone detection approach based on suffix trees has been proposed by Tairas and Gray based on Microsoft's new Phoenix framework [111].

A novel approach for detecting similar trees has been presented by Jiang et al. [52] in their tool *Deckard*. In their approach, 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 can also be classified as a metrics-based techniques) and thus finds 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 and statement that yield values that can be compared to find clones of these units. In most cases, the source code is first parsed to an AST or control flow graph (CFG) on which the metrics are then calculated. Mayrand et al. [87] 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. A function clone is identified as a pair of whole function bodies with similar metrics values. Patenaude et al. use very similar method-level metrics to extend the Bell Canada Datrix tool to find Java clones [93].

Kontogiannis et al. [66] have proposed two different ways of detecting clones. One approach uses direct comparison of metrics values as a surrogate for similarity at the granularity of $begin - end$ blocks. Five well known metrics that capture data and control flow properties are 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. [10] in their tool SMC (similar methods classifier), using a hybrid approach that combines characterization metrics with dynamic matching.

Davey et al. [31] 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 [23, 38].

4.4. Semantic Approaches

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

In some approaches, the program is represented as a program dependency graph (PDG). The nodes of this graph represent expressions and statements, while the edges represent control and data dependencies. This representation abstracts from the lexical order in which expressions and statements occur to the extent that they are semantically independent. The search for clones is then turned into the problem of finding isomporphic subgraphs (for which only approximate efficient algorithms exist) [65, 75, 81]. One of the leading PDG-based clone detection tools is proposed by Komondoor and Horwitz [65], which finds isomorphic PDG subgraphs using (backward) program slicing. Krinke [75] uses an iterative approach (k-length patch matching) for detecting maximally similar subgraphs in the PDG. Liu et al. [81] have developed a plagiarism detector based on PDGs. Another recent study by Gabel et al. [46] maps PDG subgraphs to related structured syntax and then finds clones using *Deckard*.

4.5. Hybrids

In addition to the above, there are also clone detection techniques that use a combination of syntactic and semantic characteristics. Leitao [79] provides a hybrid approach that combines

syntactic techniques based on AST metrics and semantic techniques (using call graphs) in combination with specialized comparison functions.

5. Comparison of Tools

Clone detection tools are multivariate, and therefore their study requires a systematic scheme for describing their properties. In this comparison, we will describe the properties of clone detection tools according to such a systematic classification. Our classification scheme is outlined first, and then we classify and compare the techniques and tools using it.

The properties are organized into *facets*, each of which may have different, but not necessarily disjoint *attribute values*. Related facets are grouped into *categories*. We first introduce the categories, facets, and attributes and then classify the tools and techniques in this scheme.

In order to provide a 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. While the citations for the different facets and attributes in Tables 2 to 11 may not be complete , we provide the values for all facets and attributes of the individual tools and techniques in Table 12.

5.1. Usage Facets

The category *Usage* groups facets relevant to the usage of a technique or tool. Table 2 lists the usage facets. The second column in the table gives the full name of the facet, and the first column gives the mnemonic abbreviation we use to refer to it. Unique identifiers for the facet's attribute values are found in the third column. The last column gives short descriptions of the attribute values along with the citations of the corresponding techniques and tools.

Platform: The facet *Platform* describes the execution platform for which the tool is available.

External Dependencies: The *External Dependencies* facet states whether the tool requires a special environment or additional other tools to work.

Availability: The *Availability* facet describes the kind of license under which the tool is made available.

5.2. Interaction Facets

The interaction category deals with how a user interacts with the clone detection tool (cf. Table 3), an important consideration when adopting a tool.

User Interface: This facet describes whether the tool supports interactivity or whether it is used in batch mode.

Output: The *Output* facet indicates the kind of output supported by the particular tool. Some tools provide cloning information textually with file name and begin-end line numbers of the cloned fragments, some provide the original source of the cloned fragments in a suitable format, some show the abstracted view of the cloned code (e.g., scatter-plot view) and some provide a combination of these.

IDE Support: The *Plug-in Support* facet indicates whether the tool is part of an integrated development environment (IDE). Only a few tools provide direct IDE support.

Table 2: Usage Facets

Abb.	Facet	Attr.	Description
P	Platform	P.a	The tool is platform independent T [78], L [29], S [20]
		P.b	The tool has been run on Linux/Unix T [104, 99], L [8], S [52], M [93]
		P.c	The tool has been run on Windows L [12, 76, 37, 115], LS [74], S [111, 113], G [65, 81, 46]
		P.d	The tool has been run on both Windows and Linux/Unix T [107], L [59, 112], S [14, 72, 15, 108]
		P.e	Others / Information not available T [55, 41, 86, 85, 114, 92], L [84], S [42, 116, 57], M [66, 87, 31, 67], G [75]
D	External Dependencies	D.a	Possibly the tool has no external dependencies T [99, 41, 107, 104, 55, 85, 114, 78], L [59, 12, 112, 58, 8, 115], S [52, 116]
		D.b	The tool seems to have external dependencies or to be a part of a larger tool set T [86] (PROCSSI), T [92] (recoder), L [29] (PMD), L [84] (CloSpan), L [37] (ConQAT), S [15] (DMS [13]), S [14] (Bauhaus), [20] (CPython, ANTLR), S [111] (Microsoft Phoenix Framework), LS [74] (CodeDOM of .Net), S [113](JAML), LS [72] (Bauhaus), S [42] (JavaML and lcsc), S [108] (ANTLR), M [87, 93] (Datrix), G [75] (VALSOFT), G [65, 81, 46] (CodeSurfer), G [75] (Krinke and Snelting validation framework)
		D.c	Others / Information not available S [57], M [66, 31, 67]
A	Availability	A.a	The tool is open source T [78], L [29, 37], S [20]
		A.b	The tool is freely available for research in binary form T [41, 107], L [59, 112, 58, 37], S [108]
		A.c	The tool is commercially available S [15]
		A.d	There is a free evaluation license S [15, 14]
		A.e	Probably evaluation version is available on request T [104, 99, 114], L [8, 12, 84], S [72, 52], G [65, 81, 46, 75]
		A.f	Others / Information not available / Possibly not available T [55, 85, 86, 92], L [115], S [42, 116, 111, 113, 74, 57], M [66, 66, 31, 93, 67]

5.3. Language Facets

The language category deals with the programming languages that can be analyzed using the tool. Table 4 summarizes these facets and their attribute values.

Language Paradigm: The *Language Paradigm* facet indicates the programming paradigm targeted by the tool.

Language Support: Facet *Language Support* refines *Language Paradigm* to the set of particular languages.

5.4. Clone Information Facets

The clone information category gathers facets that characterize the kinds of clone information the tool is able to emit (cf. Table 5). The richer this information and more refined its structure, the more useful it is for further processing.

Clone Relation: The *Clone Relation* facet concerns 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 for refactoring. Techniques that provide clone classes directly

Table 3: Interaction Facets

Abb.	Facet	Attr.	Description
U	User Interface	U.a	May be used as command line tool T [104, 114, 107], L [59], S [20]
		U.b	Provides a graphical user interface T [78, 92], L [115] (Clone List and File Info Views), L [112, 37], S [42], M [31]
		U.c	Both command line tool and graphical user interface (U.a) & (U.b) L [58, 59, 29, 14], S [15, 72, 108]
		U.d	Not precisely mentioned: *See Table 12 for the remaining list*
O	Nature of Output	O.a	Emits results textually providing only the source coordinates of the cloned fragments (e.g., file name and begin-end line numbers of the cloned fragments) T [85, 86, 107], L [12], S [52, 72], M [24, 66, 67]
		O.b	Emits results graphically providing the original source of the cloned fragments in a suitable format (e.g., HTML) or provides overall abstracted visual representation (e.g., dot-plot). T [25, 78, 92], L [28, 112, 115, 37], S [116, 42, 20, 47, 111], M [23]
		O.c	Both textual source coordinates of the cloned fragments and original source in suitable format or abstracted visual representation (both O.a and O.b) T [41, 104, 99, 114, 54, 55], L [58, 59, 8, 29, 14], S [15, 20, 108, 14, 16]
		O.d	Not precisely mentioned: *See Table 12 for the remaining list*
I	IDE Support	I.a	Is a Plug-in for Eclipse T [78, 39, 108, 51], S [20]
		I.b	Integrated/Dependent in other IDE S [111] (MS Phoenix framework), L [115, 76] (Visual Studio 2005), [108] (several IDEs)
		I.c	Others: *All other tools (Table 12) except listed here possibly have no IDE support*

(e.g., *RTF* [12]) may therefore be better for maintenance than those that return only clone pairs (e.g., *Dup* [8]) or require post-processing to group clones into classes (e.g., *CCFinder* [59]).

Clone Granularity: The facet *Clone Granularity* indicates the granularity of the returned clones – *free* (i.e., no syntactic boundaries), *fixed* (i.e., within predefined syntactic boundaries such as method or block) or both. Both granularities have advantages and disadvantages. For example, techniques that return only function clones are useful 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.

Clone Type: The *Clone Type* facet considers the types of clones that a technique can detect. While all techniques can detect exact clones, only few tools (e.g., *Dup* [8]) can find parameterized *Type-2* clones. This issue is discussed in detail in the context of edit-based scenarios later in the paper.

5.5. Technical Aspect Facets

The technical aspect facets category relates to the comparison algorithms, their complexity, and their unit of comparison (cf. Table 6).

Comparison Algorithm: The *Comparison Algorithm* facet identifies the different algorithms used in clone detection. For example, the suffix-tree algorithm finds all equal subsequences in a sequence composed of a fixed alphabet (e.g., characters, tokens, hash values of lines) in linear

Table 4: Language Facets

Abb.	Facet	Attr.	Description
LP	Language Paradigm	LP.a	Applied to only procedural languages T [55, 54, 85, 86, 104], S [116, 47, 111], M [24, 31, 66, 67, 87], G [65, 75]
		LP.b	Applied to only object-oriented languages T [92], L [37], S [42, 20, 113, 74, 57, 108], M [93, 10],
		LP.c	Applied to both procedural and object-oriented languages T [41, 78, 99, 114, 107], L [8, 59, 12, 58, 112, 84, 115, 29], S [15, 52, 14, 43], G [81, 46]
		LP.d	Applied to web languages T [107], [38, 36, 35, 95, 28, 23, 94, 77, 62, 49]
		LP.e	Applied to only functional languages [80]
		LP.f	Applied to modeling languages L [82] (Sequence Diagram), G [33] (Simulink)
		LP.g	Applied to Lisp-like languages SMG [79] (hybrid)
		LP.h	Applied to assembler code [26, 34, 32, 45]
		LP.i	Applied to Java Byte Code [5]
		LP.j	Applicable across different languages LS [74] (currently C# and Visual Basic.NET)
LS	Language Support	LS.a	Is language independent T [78], T [107] (has several other language-specific lexical options too) L [37] (has several other language-specific lexical options too)
		LS.b	Experimented with "C" T [55, 41, 114, 54, 78, 86, 85, 104, 99], L [59, 8, 12, 58, 112, 84, 29], S [15, 116, 47, 52, 72, 14, 111], M [87, 66, 24, 31, 67], G [65, 75, 81, 46]
		LS.c	Experimented with "C++" T [41], L [59, 58, 112, 84, 115, 29], S [14, 15, 113], G [81, 46]
		LS.d	Experimented with "C#" T [97] L [115, 37], L [59, 58, 112], S [42], LS [74]
		LS.e	Experimented with "Java" T [41, 114, 78, 99, 92], L [59, 8, 12, 29], S [15, 42, 52, 14, 20, 43, 113, 57, 108], M [93, 10], G [81]
		LS.f	Experimented with "COBOL" T [41], L [59, 58, 112], S [14, 15]
		LS.g	Experimented with "Python" S [20]
		LS.h	Experimented with "HTML" L [28, 110]
		LS.i	Experimented with "Visual Basic" L [59, 58, 112, 115], S [74]

time and space, but can handle only exact sequences. On the other hand, data mining algorithms are well suited to handle arbitrary gaps in the subsequences.

Comparison Granularity: Different techniques work at different levels of comparison granularity, from single tokens and source lines to entire AST subtrees and PDG subgraphs. The facet *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 determines also 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.

Table 5: Clone Facets

Abb.	Facet	Attr.	Description
R	Clone Relation	R.a	Yields clone pairs T [41, 114, 78, 92], L [8, 59, 58, 29], S [15, 72, 14, 20, 42, 111, 74, 108], M [87, 23, 24, 66, 93, 67], G [65, 75, 81]
		R.b	Yields clone classes T [104, 99, 55, 54, 85, 86, 107] L [12, 28, 115, 84, 37], S [52, 113], M [10, 31], G [81, 33, 46]
		R.c	Yields both clone pairs and clone classes directly by the comparison algorithm (note: *None can directly find both clone pairs and clone classes.*)
		R.d	Groups clone pairs in classes in post-processing T [41], L [59, 58], S [15, 111, 108, 14], M [10], G [65]
		R.e	Others S [116, 57]
G	Clone Granularity	G.a	Free T [41, 54, 55, 114, 78, 86, 107], L [59, 58, 8, 12, 115, 29, 37], S [15, 42, 47, 52, 72, 14, 20, 113, 108], G [75, 65]
		G.b	Fixed, Function/Method T [99, 104], S [111, 74], M [87, 10, 31, 23, 24, 66, 67], G [65, 81, 46]
		G.c	Fixed, begin-end block T [99, 104], L [28], M [66]
		G.d	Fixed, any structured block T [99, 104], L [28]
		G.e.	Fixed, Class S [109]
		G.f	Fixed, File T [85], S [116]
		G.g	Others L [84] (Basic Block), S [57] (sub-statement)
CT	Clone Types	CT.a	*Type-1* or subset of *Type-1*: *All the tools/techniques listed in Table 12 can detect such clones (or a subset) with some limitations.*
		CT.b	*Type-2* or subset of *Type-2*: *Except some text-based techniques/tools [55, 41, 114, 78] and one tree-based [57], all others are able to detect such clones (or a subset) with some limitations.*
		CT.c	*Type-3* (near-miss) or subset of *Type-3*. Some techniques/tools might have some limitations T [41, 114, 78, 85, 104, 99, 55, 86], L [84, 112], S [15, 52, 42, 47, 14, 20, 108], M [87, 10, 66, 92, 23, 24], G [65, 75, 81, 46]
		CT.d	*Type-4* or subset of *Type-4*. Some techniques/tools might have some limitations T [86], G [65, 75, 81, 46]
		CT.e	Others T [86] (ADT), T [25] (Visualization only), S [116] (Visualization only)

Worst Case Computational Complexity: 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, the comparison algorithm used, and the granularity of its use. The facet *Computational Complexity* indicates the overall computational complexity of a particular technique/tool.

5.6. Adjustment Facets

The adjustments category relates to ways the tool allows a user to make adjustments to the search. Adjustments are offered by way of heuristics that may be turned on and off, thresholds that may be set, and various kinds of pre- and post-processing (cf. Table 7).

Pre-/Post-Processing: The facet *Pre-/Post-Processing* refers to any special pre- or post-processing (e.g., pretty printing) as outlined in Sections 3.1 and 3.5 that are required other than

Table 6: Technical Facets

Abb.	Facet	Attr.	Description
CA	Comparison Algorithms	CA.a	Suffix tree L [59, 8, 37], S [20, 72, 111], G [82]
		CA.b	Suffix array L [115, 12]
		CA.c	AST-based Suffixtree S [72, 111]
		CA.d	dotplot/scatter plot T [114, 25], L [112]
		CA.e	Dynamic pattern matching T [41], M [10, 24, 66]
		CA.f	Data Mining L [84] (Frequent Sub Sequence), S [113] (Frequent Itemset)
		CA.g	Information Retrieval [86] (Latent Semantic Indexing)
		CA.h	Hash-value comparison S [15, 52, 14]
		CA.i	Fingerprinting T [55, 85, 54]
		CA.j	Neural Networks M [31]
		CA.k	Graph matching G [75, 81], G [65] (slicing), G [33] (model)
		CA.l	Sub-tree matching S [15] (hashing), S [14]
		CA.m	Euclidean distance M [67, 38]
		CA.n	Levenshtein distance LS [74]
		CA.o	Other sequence matching T [78] (n-neighbor), T [104] (similar to Unix *diff*), T [92] (Edit distance), L [28] (diff), S [116, 47] (dynamic programming),
		CA.p	Hybrid SMG [79]
		CA.q	Others T [107], L [29] (Karp-Rabin string matching), S [57] (lossless data compression), S [42, 108], M [87] (discrete comparison), [93], G [46] (Locality sensitive hashing)
CU	Comparison Granularity	CU.a	Line T [41, 114, 25, 104, 107], L [28], L [8] (p-tokens of line)
		CU.b	Substring/fingerprint T [54, 55, 85] (multi-line), T [78] (multi-word)
		CU.c	Identifiers and Comments T [86]
		CU.d	Tokens L [59, 12, 8, 115, 29, 37, 14], S [72, 111] (tokens of suffix trees), S [74] (Tokens of codeDOM graph), S [57] (Tokens of ASTs)
		CU.e	Statements L [84], S [113]
		CU.f	Subtree S [15, 14, 116, 42, 52, 14, 20, 108]
		CU.g	Subgraph G [65, 75, 81]
		CU.h	Begin-End Blocks M [66]
		CU.i	Methods S [111], M [87, 10, 23, 93, 24, 66, 67]
		CU.j	Files T [85], S [116]
		CU.k	Others T [92] (Atomic instructions) L [112] (uses non-gapped clones),
CC	Worst Case Computational Complexity	CC.a	Linear T [78], L [59, 8, 12, 115, 37, 14], S [72], [72, 43]
		CC.b	Quadratic T [41, 104] (wrt. no of lines/potential clones), L [112] (wrt. no. of non-gapped clones), S [15, 14, 52, 116, 47, 111, 20, 113], M [10, 23, 24, 66, 87, 31, 93, 67],(wrt. no. of methods/begin-end blocks)
		CC.c	Polynomial G [75, 65, 81, 33]
		CC.d	Others/Not precisely defined T [55, 85, 114, 86, 92, 107], L [84, 29], S [42, 74, 57, 108]

the usual filtering of whitespace and comments with light-weight parsing or regular expressions [41, 114].

Heuristics/Thresholds: The *Heuristics/Thresholds* facet indicates whether there are any

thresholds or heuristics used by a particular technique/tool that may be manipulated by a user.

Table 7: Adjustment Facets

Abb.	Facet	Attr.	Description
PP	Pre-/Post-Processing	PP.a	Pre-processing T [114, 78], L [28, 104, 99, 41, 92], S [47, 116]
		PP.b	Post-processing T [85], L [8, 59, 112, 84, 115], S [72, 52], G [65, 46]
		PP.c	Others/Possibly none T [55, 86, 107], L [12, 29, 37, 14], S [15, 42, 111, 113, 20, 74, 57, 108, 14], M [10, 23, 24, 66, 87, 31, 93, 67], G [75, 81]
H	Heuristics/ Thresholds	H.a	On clone length T [41, 114, 14], T [78] (4 words), T [54, 55] (50 lines), L [8] (15 lines), L [59, 58, 112, 12, 115, 37] (e.g., 30 tokens), LS [72]
		H.b	On code similarity T [99, 104, 114, 86, 92], L [8, 59, 84, 12, 28, 115], S [15, 52, 14, 42, 20, 108], LS [74], M [10, 23, 24, 66, 87, 31, 93, 67], G [81, 46]
		H.c	On gap size T [41, 78, 114, 99, 104, 85], L [84, 112, 29], S [52, 42, 47, 113], M [10], G [75, 46]
		H.d	On pruning T [41, 54, 55, 114], L [59, 12]*, L [84], S [72], G [81, 46]
		H.e	Others/Possibly none T [107], S [116, 111, 57], G [65]

5.7. *Processing Facets*

The processing category includes facets that characterize the ways a tool analyzes, represents, and transforms the program for the comparison.

Basic Transformation/Normalization: Noise (e.g., comments) filtering, normalization and transformation of program elements are important steps in clone detection tools, helping both in removing uninteresting clones (filtering), and in finding near-miss clones (normalization and transformation). The *Basic Transformation/Normalization* facet deals with this issue (cf. Table 8).

Table 8: Basic Transformation/Normalization Facet

Attr.	Description
T.a	No normalization of source code T [78, 86, 85, 114] (whitespace and single brackets) M [23]
T.b	Remove comments and whitespace with regular expressions or light-weight parsing T [41, 54, 114], L [8, 12]
T.c	Remove comments and whitespace in parsing and apply some kind of pretty-printing/text-processing to remove formatting differences between similar fragments. T [104, 99]
T.d	Comments are not removed but also taken into consideration for comparison T [86, 54]*, M [87]
T.e	Apply normalizaition of identifiers, types and literal values T [107], L [59, 58, 112, 84, 115, 37]
T.f	Identifier names (and comments) are kept and compared for finding clones T [86]
T.g	There is flexible normalization of the identifiers (different options are provided to the user) T [104], L [12, 29], S [52]
T.h	Several language dependent transformation rules are applied T[104] (Example like TXL transformation rules), T [92] (Semantic preserving transformation rules to get sequence of atomic instructions), L [59] (Token transformation rules)
T.i	Comments and whitespace are ignored in parsing or while generating graphs T [104, 99], S [15, 52, 116, 42, 47, 14, 72, 111, 113], M [10, 24, 66, 67, 93], G [65, 75, 81, 46]

Code Representation: The *Code Representation* facet refers to the internal code representation after filtering, normalization and transformation (cf. Table 9). 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 4. 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* [72] 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.

Table 9: Code Representation Facet

Attr.	Description
CR.a	Raw source without any changes: *Possibly none*
CR.b	Filtered Strings: Effective lines of code after removing comments and whitespace (possibly line breaks are not removed) T [41, 114, 78, 55, 54], L [28]
CR.c	Line breaks are also removed in filtered strings: *Most token-based tools do this*
CR.d	Filtered subtrings with comments, whitespace and line breaks may or may not be removed T [54, 85] (fingerprint), L [115]
CR.e	Fingerprinting of substrings with comments, whitespace and line breaks may or may not be removed T [107]
CR.f	Normalized strings/Token sequence with comments, whitespace and line breaks may or may not be removed T [41, 40], L [59, 58, 112, 12, 115, 37, 14] (token sequence)
CR.g	Parameterized strings/Token sequence with comments, whitespace and line breaks may or may not be removed L [8, 6], (p-token sequence), L [14]
CR.h	Words in context T [86]
CR.i	Metrics/Vectors S [52] (characteristic vector), M [87] (IRL), M [66, 10, 93, 23, 24, 67, 31], G [46]
CR.j	Abstract Syntax Tree (AST) or Annotated AST or AST nodes are in another form S [15, 116, 108], S [14](IML), S [42](XML), S [47](string alignment), S [72, 111](suffix-trees)
CR.k	PDG or variants of PDG G [65, 81] (PDG), G [75] (PDG+AST)
CR.l	AST/Parse-tree is in another form S [20, 113] (XML), LS [74] (CodeDOM), S [57](Tokens of AST-nodes), M [87] (IRL)
CR.m	Pretty-printed text without comments T [99],
CR.n	Normalized/transformed Text T [104] (also pretty-printed), L [84] (Mapping statements to numbers)
CR.o	Hybrid SMG [79] (AST+Metrics+call graph)
CR.p	Others T [92] (sequence of atomic instructions), L [29] (Frequency table of tokens) G [33] (normalized graph*),

Program Analysis: The facet *Program Analysis* indicates the kind of program analysis required for a particular technique in order to produce the intermediate representation (cf. Table 10). While most text-based techniques work directly on source code and token-based techniques generally require only lexical analysis, other techniques/tools can be very language-dependent (e.g., requiring a full parser).

5.8. Evaluation Facets

Empirical validation of tools is important, especially in terms of precision, recall, and scalability. The evaluation category deals with evaluation aspects (cf. Table 11). These facets can assist

Table 10: Program Analysis Facet

Attr.	Description
PA.a	Nothing, completely language independent T [85], T [107] (has several other language-specific options too)
PA.b	Only needs some regular expressions for removing comments and whitespace or so T [114]
PA.c	Only needs lightweight parsing for removing comments, whitespace and pretty-printing (or so) of the code T [41, 54, 55, 78, 86], M [23]
PA.d	Needs a lexer at least for removing comments / whitespace and to tokenize the source L [59, 58, 112, 6, 8, 12, 115, 29, 37, 14],
PA.e	Needs a full-fledged parser or IDE to generate parse tree/AST or to find another representation of the source L [84], S [15, 116, 42, 47, 72, 14, 20, 111, 113, 74, 57, 108, 16, 14], M [87, 10, 24, 66, 67, 31, 93],
PA.f	Needs specialized tool to generate Call Graphs, traditional PDGs or annotated special PDGs G [65, 75, 81], SMG [79] (call graph)
PA.g	Needs language dependent transformation rules also T[104] (full NICAD), T [92], L [59] (lexical)
PA.h	Needs only a context-free grammar for the language dialect of interest T [99] (Basic NICAD) (in TXL), L [28] (in TXL), S [52]

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.

Empirical Validation: This facet hints at the kind of validation that has been reported for each technique.

Availability of Empirical Results: The facet *Availability of Empirical Results* notes whether the results of the validation are available. If the results are available, other researchers may be able to replicate, compare and extend them with additional studies.

Subject Systems: The *Subject Systems* facet notes which systems have been used in the validation. If researchers conduct their empirical studies on the same systems, results can be compared more meaningfully.

5.9. Tool Classification and Attributes

In this section we provide the attribute values for the facets for each of the individual tools in our study. Table 12 presents a detailed overview of the available tools and techniques in the form of a taxonomy where the first column (*Col. 1*) groups by the underlying approach, and the second column (*Col. 2*) lists each tool / technique by name (or first author name for those technique without a tool name) and citations. The third column (*Col. 3*) gives the attribute values for the *Usage* facets of Table 2 that apply to the tool / technique, and the remaining columns give the attribute values for the other facets, *Interaction*, *Language*, *Clones*, *Technical Aspects*, *Adjustments*, *Basic Transformation/Normalization*, *Code Representation*, *Program Analysis* and *Evaluation*, as described in Tables 3, 4, 5, 6, 7, 8, 9, 10 and 11 respectively.

A particular tool / technique can have multiple attribute values for a facet, represented as a sequence of attribute letters. For example, the attribute value “acg” for facet *F* refers to attributes

Table 11: Evaluation Facet

Abb.	Facet	Attr.	Description
E	Empirical Validation	E.a	Yes, validated empirically in terms of precision, recall, memory and time and compared with other tools S [72, 43]
		E.b	Validated enough in support of the claim T [41, 114, 86, 92, 104], L [8, 6, 83, 84], S [42, 15, 52], M [10, 66, 87, 67], G [81, 46]
		E.c	Validated by other means or third party comparison study T [104, 99] (with an automatic validation framework [98, 101]), T [41] (with Bellon's experiment [18]), L [6] (with Bellon's experiment [18]), L [59] (with Bellon's experiment [18]), S [15] (with Bellon's experiment [18]), M [87] (with Bellon's experiment [18]), G [75] (with Bellon's experiment [18]),
		E.d	Others (Possibly not validated exhaustively) T [78, 54, 55, 85, 107], L [12, 28, 115, 29, 37, 14], S [116, 111, 113, 74, 57, 108], M [23, 31, 93], G [65, 75]
AR	Availability of Results	AR.a	Yes, complete results T [99, 104] (see at [100]), S [20] (see at [21]), Experiment [18] (see at [17])
		AR.b	Enough/Partial results as in the published paper (or online documents) T [54, 41, 85, 114, 78, 86, 25, 92, 107], L [8, 59, 83, 112, 58, 12, 84, 115, 29, 37, 14, 72, 48], S [15, 42, 116, 72, 43, 52, 111, 113, 74, 57, 108, 14, 16, 109], M [66, 87, 31, 93, 67], G [65, 75, 46, 81]
S	Subject Systems	S.a	Linux Kernel/part (C, 3M LOC) T [99], L [12, 59, 84, 14, 72], S [52], M [24], G [46]
		S.b	JDK/Part (Java, 204K LOC) T [99, 114, 78, 107], L [59, 4, 29, 14, 72], S [52, 113, 108], M [10, 93], Experiments [18]
		S.c	SNNS (C, 115K LOC) T [99, 114], L [4, 14, 72], S [72, 4], Experiments [18]
		S.d	postsql(C, 235K LOC) T [99, 114], L [84, 16, 4, 14, 72], S [72], Experiments [18, 4], G [46]
		S.e	Apache httpd or part(C, 261K LOC) T [99, 78], L [14, 72, 84]
		S.f	FreeBSD (C, 3M LOC) L [83, 84, 59]
		S.g	Others T [41, 85, 55, 92], L [115, 37], S [15, 42, 116, 111, 20, 74, 57, 16, 14], M [66, 67, 87, 31], G [75, 65, 81]

F.a, *F.c* and *F.g*. In order to focus the comparison, we have restricted this summary comparison to methods for procedural and object-oriented languages and have not listed tools and techniques aimed at other paradigms (such as web applications) in this summary.

6. Scenario-Based Evaluation of the Techniques and Tools

Clone detection techniques are often inadequately evaluated, and only a few studies have looked at some of the techniques and tools [18, 105, 106, 22]. Of these, the Bellon et al. [18] 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 C and Java. However, even in that careful study, only a small proportion of the clones were oracled, and a number of other factors have been identified as potentially influencing the results [4]. The general lack of evaluation is exacerbated by the fact that there are no agreed upon evaluation criteria or representative benchmarks. Finding

Table 12: Tools Attributes

Col. 1	Col. 2	Col. 3			Col. 4			Col. 5		Col. 6			Col. 7			Col. 8		Col. 9			Col. 10		
		Usage (Table 2)			Interaction (Table 3)			Language (Table 4)		Clones (Table 5)			Technical Aspects (Table 6)			Adjustments (Table 7)		Processing (Section 5.7)			Evaluation (Table 11)		
Approach	Tool/Author	P Platform	D External Dependencies	A Availability	U User Interface	O Output	I IDE Support	LP Language Paradigm	LS Language Support	R Clone Relation	G Clone Granularity	CT Clone Types	CA Comparison Algorithms	CU Comparison Granularity	CC Computational Complexity	PP Pre-/Post-Processing	H Heuristics/Thresholds	T Transformations (Table 8)	CR Code Representation (Table 9)	PA Program Analysis (Table 10)	E Empirical Validation	AR Availability of Results	S Subject Systems
Text-Based	Johnson [55, 54, 53]	e	a	f	d	c	c	a	b	b	a	a	i	b	d	c	ad	cd	bd	c	d	b	g
	Duploc [41]	e	a	b	d	c	c	c	bcef	ad	a	ac	e	a	b	a	cd	b	bf	c	bc	b	g
	sif* [85]	e	a	f	d	a	c	a	b	b	f	ac	i	b	d	b	c	a	d	a	d	b	g
	DuDe [114]	e	a	e	a	c	c	c	be	a	a	ac	d	a	d	a	abc	ab	b	b	d	b	bcd
	SDD [78]	a	a	a	b	b	a	c	abe	a	a	ac	o	b	a	a	ac	a	b	c	d	b	be
	Marcus* [86]	e	b	f	d	a	c	a	b	b	a	acde	g	c	d	c	b	adf	h	c	b	b	g
	NICAD [99, 104]	b	a	e	a	c	c	c	bde	b	bcd	abcd	o	a	b	a	bc	cghi	n	g	bc	a	abcde
	Nasehi [92]	e	b	f	b	b	c	b	e	a	b	abcd	o	k	d	a	b	h	p	g	d	b	g
	Simian [107]	d	a	b	a	a	c	cd	a	b	a	ab	q	a	d	c	e	e	f	a	d	b	b
Token-Based	Dup [8]	b	a	e	d	c	c	c	be	a	a	ab	a	ad	a	b	ab	b	g	d	bc	b	bcd
	CCFinder(X) [59, 58]	d	a	b	c	c	c	c	bcef	ad	a	abc	a	d	a	b	abd	eh	f	dg	bc	b	abf
	RTF [12]	c	a	e	d	a	c	c	be	b	a	ab	b	d	a	c	abd	bg	f	d	d	b	a
	CP-Miner [84]	e	b	e	d	d	c	c	bc	b	g	abc	f	e	d	b	cd	e	n	e	b	b	adef
	SHINOBI* [115]	c	a	f	b	b	b	c	cdi	b	a	ab	b	d	a	b	ab	e	d	l	d	b	g
	CPD [29]	a	b	a	c	c	c	c	bce	a	a	ab	q	d	d	c	c	g	p	d	d	b	b
	Clone Detective [37]	c	b	ab	b	b	b	b	ad	b	a	ab	q	d	a	c	a	e	f	d	d	b	g
	clones [14, 72]	d	b	d	c	c	c	c	bdefi	ad	a	ab	a	ad	a	c	a	ah	fg	d	b	b	abcde
Tree-Based	CloneDr [15]	c	b	cd	c	c	c	c	bcef	ad	a	abc	hl	f	b	c	b	i	j	e	bc	b	g
	Asta [42]	e	b	f	b	b	c	b	de	a	a	abc	q	f	d	c	bc	hi	j	e	b	b	g
	cdiff* [116]	e	a	f	d	b	c	a	b	e	f	abe	o	f	b	a	e	i	j	e	d	b	g
	cpdetector [72, 43]	d	b	e	c	a	c	c	be	a	a	ab	a	d	a	b	ad	i	j	e	a	b	cd
	Deckard [52]	b	a	e	d	a	c	c	be	b	a	abc	h	f	b	b	bc	i	i	h	b	b	ab
	Tairas [111]	c	b	f	d	b	b	a	b	ad	b	ab	a	d	b	c	e	i	j	e	d	b	g
	CloneDetection [113]	c	b	f	d	d	c	b	ce	b	a	ab	f	e	b	c	c	i	l	e	d	b	b
	CloneDigger [20]	a	b	a	a	c	a	b	eg	a	a	abc	a	f	b	c	b	i	l	e	d	a	g
	C2D2 [74]	c	b	f	d	d	c	b	di	a	b	abc	n	d	d	c	ab	i	l	e	d	b	g
	Juillerat [57]	e	c	f	d	d	c	b	e	e	g	a	q	d	d	c	e	i	l	e	d	b	g
	SimScan [108]	d	b	b	c	c	ab	b	e	ad	a	abc	q	f	d	c	b	i	j	e	d	b	b
	ccdiml [16, 14]	d	b	d	c	c	c	c	bcef	a	a	abc	l	f	b	c	b	i	j	e	b	b	g
Metrics-Based	Kontogiannis [66]	e	c	f	d	d	c	a	b	a	bc	abcd	e	hi	b	c	b	i	i	e	b	b	g
	Mayrand [87]	e	b	f	d	d	c	a	b	a	b	abcd	q	i	b	c	b	d	i	e	bc	b	g
	Davey [31]	e	c	f	b	b	c	a	b	b	b	abcd	j	i	b	c	b	d	i	e	d	b	g
	Patenaude [93]	b	b	f	d	d	c	b	e	a	b	abcd	q	i	b	c	b	i	i	e	d	b	b
	Kontogiannis [67]	e	c	f	d	d	c	a	b	a	b	abcd	m	i	b	c	b	i	i	e	b	b	g
Graph-Based	Duplix [75]	e	b	e	d	d	c	a	b	a	a	abcd	k	g	c	c	c	i	k	f	cd	b	g
	Komondoor [65]	c	b	e	d	d	c	a	b	ad	ab	abcd	k	g	c	b	e	i	k	f	d	b	g
	GPLAG* [81]	c	b	e	d	d	c	c	bce	a	b	abcd	k	g	c	c	bd	i	k	f	b	b	g
	Gabel [46]	c	b	e	d	d	c	c	bc	b	b	abcd	q	i	c	b	bcd	i	i	f	b	b	ad

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 in the form of a top-down editing taxonomy. Deriving such scenarios is itself challenging, since the definition of clones is inherently vague in the literature [102, 73]. Baxter et al. [15] give the most general definition, defining clones simply as segments of code that are similar according to some definition of similarity. Kamiya et al. [59] 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. A similar definition is used by Burd et al. [22], 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. [15], have defined "similar" using detection-dependent definitions in terms of difference thresholds [60, 67, 84], and it has been proposed that automatically combining multiple detector result sets can help overcome such similarity definition problems [18, 67]. Categorization in the form of clone taxonomies has been suggested as a way to avoid such ambiguities in definition [10, 87]. However, these taxonomies are limited to function clones and still use vague terms such as "similar" [87] and "one/two/three long difference" [10].

Intuitively, in most cases the "clones" we are looking for are those created as a result of copy-/paste/modify actions by programmers. In our work we begin with this assumption, and use it as the basis of a top-down theory of clones, which we have formalized into a taxonomy of editing scenarios that a programmer may undertake in the intentional creation of a clone. Our taxonomy is not simply guesswork - it is derived from the large body of published work on existing clone definitions [15, 46, 59, 65, 84], clone types [18, 67], clone taxonomies [10, 60, 87], a study of developer copy/paste activities [63] and other empirical studies [3, 11, 61, 64]. We have validated the taxonomy by studying the copy/paste patterns of function clones [100] from an empirical study that analyzed 17 open source C and Java systems including the entire Linux Kernel (6,265 KLOC C, 154,977 functions), Apache *httpd* (275 KLOC C, 4,301 functions) and *j2sdk-swing* (204 KLOC Java, 10,971 methods) [99].

Figure 2 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 (in the middle, labeled "Original Copy") 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. We assume that our primary intention is to find true clones, that is, those that actually result from copy-and-edit reuse of code. Figure 2 shows four scenarios, *Scenario 1*, *Scenario 2*, *Scenario 3* and *Scenario 4*, where each scenario has several sub-scenarios. Mapping to the literature (Section 2), we call the clones created by these scenarios *Type-1*, *Type-2*, *Type-3* and *Type-4* clones respectively.

From a program comprehension point of view, finding such true clones (those are created as per the scenarios) is useful since understanding a representative copy from a clone group assists in understanding all copies in that group [54]. Moreover, replacing all the detected similar copies of a

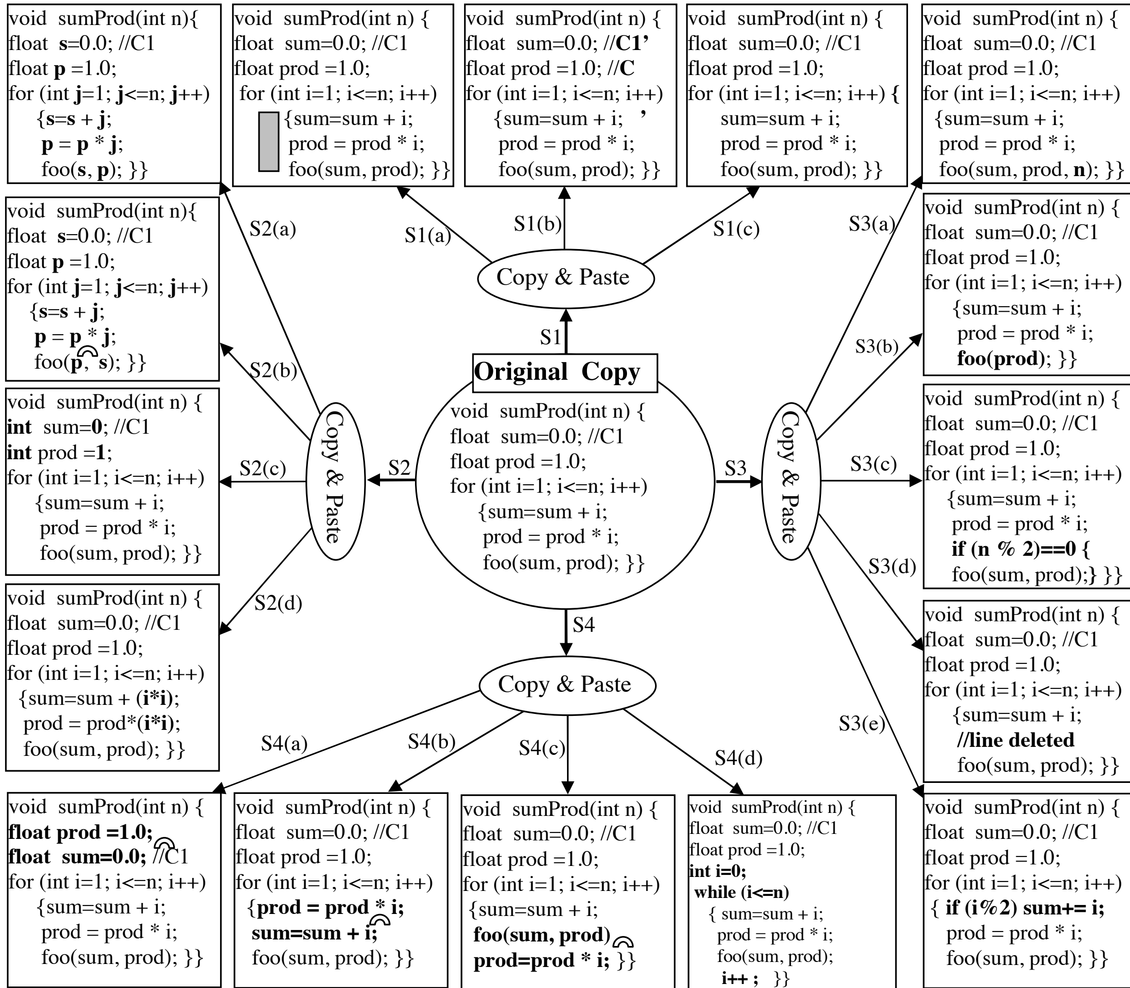

Figure 2: Taxonomy of Editing Scenarios for Different Clone Types

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 [44]. These scenarios could also be used to guide the development of forward clone management tools (e.g., *CReN* [51]).

Based on these hypothetical scenarios, we have estimated how well the various clone detection techniques may perform based on their *published* properties (either in the corresponding published papers or online documentation). In order to estimate maximal potential, we have assumed the most lenient settings of any tunable parameters of the techniques/tools. Thus, this is not an actual evaluation, rather it provides an overall picture of the potential of each technique and tool 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 and qualitative if it were cast as one, bound to target languages, platforms and implementations.

Table 14 provides an overall summary of the results of our evaluations, where the symbols

represent an estimate of the ability of each technique/tool to accurately detect each (sub-) scenario with both high precision and high recall. For example, a *very well* (denoted with ●) rating for a particular sub-scenario of a particular tool means that the subject tool (or the corresponding technique used in that tool) is capable of detecting (i.e., about 100% recall) that scenario without any false positives (i.e., about 100% precision) as per our understanding. When a tool's tunable parameters are set to detect a sub-scenario of a particular scenario, detection of the other sub-scenarios of that scenario is not counted as false positives.

However, detecting the sub-scenarios of other scenarios may be considered as false positives. Because the taxonomy is created as a top-down theory of clones from *Scenario 1* to *Scenario 4*, when a tool is set to detect a sub-scenario of a lower numbered scenario (e.g., *Scenario 1*), any detection of sub-scenario(s) of a higher numbered scenario (i.e., scenarios 2, 3 or 4) is considered as a false positive. On the other hand, when the tool is set to detect a sub-scenario of a higher numbered scenario (e.g., *Scenario 3*), the detection of the sub-scenarios of the lower numbered scenarios (i.e., scenarios 1 and 2) is desirable (for high recall) and should not be considered as false positives. Table 13 summarizes the meanings of the symbols we have used in the evaluation.

For *Scenario 2*, we also expect that a tool may provide different tunable parameters to detect the different sub-scenarios separately. For example, there may be a separate option for detecting consistently renamed clones from renaming where consistency is not maintained. This is important because some tools use the same technique but differ with respect to the tunable parameters for different types of clones. Moreover, some tools yield syntactic clones while others do not, leading to the lower ratings for a (sub)-scenario.

Given the fact that not all tools actually produce the expected output in practice, we have also employed our experience in tool comparison and knowledge gained from other tool comparison experiments and individual tool evaluations where applicable. Thus, the ratings in Table 14 for each (sub)-scenario is a balance of what is expected and what is achieved (where applicable) using a particular tool, potentially hinting the overall ability of the tool with respect to the (sub)-scenarios. Although the scenarios are represented in the language C, when we evaluate a tool that supports only object-oriented languages (e.g., Java), we imagine similar scenarios on that language to evaluate the tool (the adaptability/portability is a separate issue and discussed in the previous section).

An asterisk (*) indicates a technique/tool with special limitations (or that has some other main purpose other than clone detection) such as whole file comparison, visualization only, plagiarism detection, IDE support or other special issues discussed as applicable. In the following subsections, we consider each scenario and outline our reasoning in estimating the ability of the techniques to accurately detect them using the information from Section 5.

6.1. Scenario 1

Scenario 1: *A programmer copies a function that calculates the sum and product of a loop variable and calls another function,* foo() *with these values as parameters 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 2).*

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 along with the original. The third

Table 13: Meanings of the rating symbols

Symbols	Meaning	Description
●	very well	Detects the clones with high accuracy and confidence, i.e., with high precision and recall.
		Has tunable parameters for different types of clones (i.e., can detect clones of different scenarios separately).
		In case of *Scenario 2*, has separate tunable parameters for detecting clones of the sub-scenarios.
		When detecting clones of a sub-scenario of scenario *k* (except for *Scenario 2*), detection of the clones of other sub-scenarios of *k* is desirable for high recall.
		The scenarios are on a top-down fashion and thus, when detecting clones of scenario *k*, detection of clones of (sub-)scenario *l* where $k < l$ is not expected (for high precision). However, detection of clones of (sub-)scenario *j* where $j < k$ is desirable.
		The tool either has an option for detecting different granularities (e.g., method or begin-end block) of clones or applies several pre-/post-processing activities to avoid spurious clones [72] or at least (if the tool finds clones of free granularity) subsumes the clones of the (sub-)scenario in question.
		The tool is capable of detecting the clones of the (sub)-scenario with reasonable time and space (not in months for example)
		To our knowledge there is no empirical studies that shows that the subject tool was not capable (or performed poorly) of detecting the clone type in question.
◓	well	Detects the clones of the (sub-)scenario but may return few false positives.
		May also miss some of the clones.
		Does not meet one or more of the criteria of the first row (for *very well*).
◒	medium	Detects the clones of the (sub-)scenario but may return many false positives (about 50% for example).
		Does not meet many of the criteria of the first row (for *very well*).
⊖	low	Detects with lots of false positives (low precision).
		Also may miss many of the similar clones (low recall).
		Does not meet many of the criteria of the first row (for *very well*).
⊙	probably can	Although there is no empirical or other sort of evidence, the underlying technique of the technique/tool might be capable of detecting clones of the (sub-)scenario in question.
		The tool/technique might generate lots of false positives (very low precision).
		The tool/technique might miss some clones (very low recall).
○	probably cannot	We are not sure but as per the underlying technique of the subject technique/tool, it might be impossible to detect the clones of the (sub-)scenario in question.
		We do not think there are empirical studies or any sort of evidence that shows that the subject tool is capable of detecting the clones of the (sub-)scenario in question.
◦	cannot	As per the underlying technique of the subject technique/tool, it is impossible to detect the clones of the (sub-)scenario in question.
		There is no empirical study or any sort of evidence that the subject tool was capable of detecting the clones of the (sub-)scenario in question.

column under the *Scenario 1* heading of Table 14 summarizes how well each technique is likely to work in these scenarios.

Among the text-based techniques and tools, only NICAD [99, 104] is expected to do very well on all the sub-scenarios, in part because it was designed with them in mind. NICAD applies a standard pretty-printing normalization that removes comments (scenario S1(b)) and formatting differences (scenario S1(c)), and uses a whitespace insensitive (Scenario S1(a)) text line-wise comparison to find clones. Although, linear in space and scalable [99], NICAD has a quadratic time complexity with respect to the number of extracted code fragments for comparison. Moreover, NICAD is parser-based and thus language specific. While adapting to a new language, one at least needs to get a TXL [27] grammar for that language. Other text-based tools, such as *Duploc* [41], *DuDe* [114] and *Simian* [107] also detect scenarios S1(a) and S1(b) very well. Unlike NICAD, *Duploc* does not rely on robust parsing – instead it uses lightweight lexical analysis to remove comments (scenario S1(b)) and whitespace (scenario S1(a)) within lines and detects clones using string-based dynamic pattern matching. *DuDe* and *Simian* do similar things by applying regular expressions (i.e., lexical analysis again). However, all of these line-based techniques / tools are sensitive to format alterations and thus may not detect scenario S1(c). Marcus's text-based LSI approach [86] is not designed to detect scenario S1(b), since it compares comments (and identifiers) in finding clones. Among the other text-based techniques, Johnson's approach [53, 54, 55] should detect all three of these sub-scenarios well. Johnson applies several options for keeping/removing whitespace and comments (thus, scenarios S1(a) and S1(b) might be detected well) and uses fingerprints of substrings for finding clones (thus might not be affected by formatting, leading to detect scenario S1(c)). SDD [78] applies n-neighbor approach (i.e., allows gaps in similarity) and thus might detect these sub-scenarios too. However, allowing gaps might lead to detect false positive clones even for these exact clones.

Among the token-based techniques/tools, *RTF* [12] and *clones* [72] should detect all three *Scenario 1* sub-scenarios well. *RTF* applies flexible tokenization and *clones* has a post-processor that can distinguish different types of clones by comparing identifier values and can differentiate other similar scenarios (e.g., sub-scenarios of *Scenario 2*). However, *clones* has problems if superflous brackets are added in the copied fragment as it compares only the sequence of tokens and does not remove brackets before comparison. Token-based techniques and tools (e.g., *CCFinder*) in general cannot differentiate between clones of *Scenario 1* and *Scenario 2*. Moreover, these techniques often return non-syntactic and spurious clones [72]. Baker's *Dup* can also detect clones of scenarios S1(a) and S1(b) very well but cannot detect clones of scenario S1(c), since *Dup* summarizes all tokens of a line at a time and thus is sensitive to formatting changes. Most other token-based techniques are not sensitive to formatting changes since they compare token-by-token.

Tree-based techniques (e.g., *cpdetector*) ignore formatting differences and comments and should detect all *Scenario 1* sub-scenarios very well if they look for exact subtrees without ignoring tree-leaves (in most cases they ignore leaves and thus a post-processing step is required to distinguish clones of *Scenario 1* and *Scenario 2*). However, some tree-based techniques use alternative representations of the parse-tree/AST (e.g., *Deckard* works on characteristic vectors of the parse tree) and may not detect them accurately (a post-processing step is required to differentiate them). Moreover, a recent study [104] shows that an AST-based exact matching function clone detection technique [111] can even miss some exact function clones.

Table 14: Scenario-Based Evaluation of the Surveyed Clone Detection Techniques and Tools

● *very well* ◓ *well* ◒ *medium* ⊖ *low* ⊙ *probably can* ◯ *probably cannot* ∘ *cannot*

	Citation	Scenario 1			Scenario 2				Scenario 3					Scenario 4			
		a	b	c	a	b	c	d	a	b	c	d	e	a	b	c	d
Text-Based	Johnson [55, 54]	●	●	◓	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘
	Duploc [41]	●	●	∘	∘	∘	∘	∘	◓	◓	∘	∘	◒	∘	∘	∘	∘
	sif [85]*	◒	◒	◒	∘	∘	⊙	⊙	⊙	⊙	∘	∘	∘	⊙	⊙	⊙	∘
	DuDe [114]	●	●	⊙	∘	∘	∘	∘	◓	◓	◒	◒	◒	∘	∘	∘	∘
	SDD [78]	◓	◓	◒	∘	∘	∘	∘	◓	◓	⊖	⊖	⊖	∘	∘	∘	∘
	Marcus [86]*	◒	∘	◒	∘	∘	◒	⊖	◒	◒	∘	∘	∘	◒	◒	◒	∘
	Basic NICAD [99]	●	●	●	⊙	⊙	⊙	⊙	◓	◓	◓	◓	◓	⊙	⊙	⊙	∘
	Full NICAD [104]	●	●	●	●	◓	◓	◓	◓	◓	◓	◓	◓	⊙	⊙	⊙	∘
	Nasehi [92]	◒	◒	◒	◒	◒	◒	◒	◒	◒	⊖	⊖	⊖	◯	◯	◯	◒
	Simian [107]	●	●	◯	◒	◒	◒	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘
Token-Based	Dup [8]	●	●	∘	●	∘	◓	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘
	CCFinder(X) [59, 58]	◓	◓	◓	◓	◓	◓	∘	⊙	⊙	∘	∘	∘	∘	∘	∘	∘
	Gemini [112]*	◓	◓	◓	◓	◒	◓	∘	◒	◒	◒	◒	◒	⊙	⊙	⊙	∘
	RTF [12]	●	●	●	◓	◓	◓	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘
	CP-Miner [84]	◓	◓	◓	◓	◓	◓	⊖	◓	◓	◒	◒	◒	⊙	⊙	⊙	∘
	SHINOBI [115]*	◓	◓	◓	◓	◓	◓	∘	⊙	⊙	∘	∘	∘	∘	∘	∘	∘
	CPD [29]	●	◯	●	◒	◒	◒	◯	∘	∘	∘	∘	∘	∘	∘	∘	∘
	Clone Detective [37]	◓	◓	◓	◓	◓	◓	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘
	clones/iClones [14, 72]	●	●	◓	●	◓	◓	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘
Tree-Based	CloneDr [15]	●	●	●	●	●	●	⊖	◒	◒	⊖	◒	◒	⊙	∘	∘	∘
	Asta [42]	●	●	●	◒	◒	◒	◒	◒	◒	∘	∘	∘	∘	∘	∘	∘
	cpdetector/clast [72, 14]	●	●	●	◓	◓	◓	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘
	Deckard [52]	◓	◓	◓	◓	◓	◓	∘	◒	◒	◒	◒	◒	∘	∘	∘	∘
	Tairas [111]	◓	◓	◓	◯	◯	◒	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘
	CloneDetection [113]	◓	◓	◓	◓	◓	◓	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘
	CloneDigger [20]	●	●	●	◓	◓	◓	◓	◓	◓	∘	∘	◯	∘	∘	∘	∘
	C2D2 [74]	◒	◒	◒	◒	◒	◒	◒	◒	◒	◯	◯	◯	∘	∘	∘	∘
	Juillerat [57]	●	●	●	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘
	SimScan [108]	●	●	●	◓	◓	◓	⊙	⊙	⊙	⊙	⊙	⊙	∘	∘	∘	∘
	ccdiml [16, 14]	●	●	●	●	●	●	∘	◒	◒	⊖	●	◒	∘	∘	∘	∘
Metrics-Based	Kontogiannis [66]	◓	◓	◓	◓	◓	◓	⊖	⊖	⊖	⊖	⊖	⊖	◒	◒	◒	⊖
	Mayrand [87]	◓	∘	◓	◓	◓	◓	◒	◒	◒	⊖	⊖	⊖	◒	◒	⊖	∘
	Dagenais [30]*	◒	◒	◒	◒	◒	◒	◒	◒	◒	⊙	⊙	⊙	◒	◒	⊙	◯
	Merlo [89, 90]	◓	◓	◓	◓	◓	◓	⊖	◒	◒	⊖	⊖	⊖	◒	◒	⊖	∘
	Davey [31]	◓	◓	◓	◓	◓	◓	⊖	◒	◒	⊙	⊙	⊙	◒	⊖	⊖	◯
	Patenaude [93]	◓	◓	◓	◓	◓	◓	⊖	⊖	⊖	⊖	⊖	⊖	⊙	⊙	⊙	∘
	Kontogiannis [67]	◓	◓	◓	◓	◓	◓	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖
	Antoniol [1, 2]	◓	◓	◓	◓	◓	◓	◒	◒	◒	⊖	⊖	⊖	◒	◒	⊖	∘
Graph-Based	Duplix [75]	◓	◓	◓	◓	◓	◓	◒	◒	◒	◒	◒	◒	◓	◒	⊖	∘
	Komondoor [65]	◓	◓	◓	◓	◓	◓	⊖	⊖	⊙	⊖	⊙	⊙	◓	◓	⊖	∘
	GPLAG [81]*	◓	◓	◓	◓	◓	◓	⊖	⊖	⊖	◒	◒	◒	◓	◓	⊖	◒
	Gabel [46]	◓	◓	◓	◓	◓	◓	◓	◓	◓	◒	◒	◒	◒	◒	◒	∘

Metrics-based techniques may return the same metrics values for other scenarios of our taxonomy and for other different fragments and thus may return false positives in our sense. Among the metrics-based approaches, Mayrand et al. [87] provide a fine-grained set of metrics for detecting function clones (and possibly also clones of *begin-end* blocks). Others (e.g., Antoniol et al. [2, 1] and Merlo et al. [90, 89]) also provide similar metrics with some minor differences and are expected to do well on these scenarios.

In theory, graph-based techniques should be good at all *Scenario 1* sub-scenarios. However, in practice they yield many variants of the actual clone pairs and that there might be similar graphs for dissimilar code blocks, reducing precision. Thus in our view they do not do well on these scenarios. However, a new variant of the *Deckard* tool maps PDG subgraphs to related structured syntax before comparison and thus might do well [46].

6.2. Scenario 2

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 consistent) in the third fragment (S2(c)), and replacing some parameters with expressions in the fourth fragment (S2(d)) (Figure 2).*

Once again, an ideal clone detection technique should detect all four modified fragments as clone pairs with the original or should form a clone class for them along with the original. Needless to say, code fragments created from *Scenario 1* might also form clone pairs or a clone class with the code fragments of this scenario. The fourth column under the *Scenario 2* heading of Table 14 summarizes how well each technique may work on these scenarios.

Text-based techniques and tools are not good at detecting clones created by these (sub)-scenarios. For detecting such scenarios token normalization / abstraction / transformation is required to remove the differences between differing identifiers and literals. Of the text-based techniques, only NICAD [104], Nasehi's approach [92] and Simian [107] can detect such scenarios (although *Simian* cannot detect scenario S2(d)). NICAD can detect consistently renamed clones (scenario S2(a)) and other renamed clones (scenarios S2(b) and S2(c)) efficiently, and using flexible code normalization thus can detect scenario S2(d) as well. An extended version of *Duploc* [40] can also detect scenarios S2(b) and S2(c), but not S2(a) and S2(d). However, although these tools (i.e., NICAD, Nasehi's approach or *Simian*) find clones by textual comparison, they actually use source transformations (in NICAD's case, code abstraction and in Nasehi's approach, a transformation of program code to atomic units) and thus a syntactic / semantic analysis is required that may not be easily adaptable to other languages. The remaining text-based techniques cannot do well with these scenarios since they normally compare program text without normalization or transformation and are therefore fragile to identifier renaming.

Token-based techniques/tools are well suited to detecting clones created by *Scenario 2*. Almost all token-based techniques and tools can detect scenarios S2(a), S2(b) and S2(c) well, but are likely to also have many false positives due to their identifier and literal normalizations (or abstractions) and the detection of spurious clones [72]. However, only *Dup* [6] and *clones/iClones* [48] are rated to be robust in detecting consistently parameter-substituted clones (scenario S2(a)) because of their use of parameterized suffix trees. Most of the tools (except *Dup*, *RTF* [12] and *clones/iClones*)

cannot differentiate between *Type-1* (clones of *Scenario 1*) and *Type-2* (clones of *Scenario 2*). *RTF* and *clones/iClones* can also differentiate between the sub-scenarios of *Scenario 2*. None of the token-based techniques (except possibly *CP-Miner* [84] that allows arbitrary gaps in comparison) can detect clones of scenario S2(d) because they neither apply structural abstraction to the program code nor allow gaps in their comparison.

With the exception of Juillerat's approach [57], which detects only exact clones, and Tairas's approach [111], which detects exact clones and a small subset of *Type-2* clones, almost all tree-based techniques may also detect scenarios S2(a), S2(b) and S2(c) very well, because these techniques normally ignore identifiers and literals when comparing. However, like some of the token-based approaches, some syntactic tools do not differentiate between clones of *Type-1* and *Type-2*. The tools *CloneDr* [15], *ccdiml* [14], *cpdetector* [72] and *clast* [14] are known to differentiate these types. For scenario S2(d), the tree-based tools *Asta* [42] and *CloneDigger* [20] seem to be well suited, as they can apply structural abstraction to arbitrary subtrees.

Metrics-and graph-based techniques can also detect these scenarios, but metrics-based approaches may return many false positives because our other scenarios can yield similar metrics values. Graph-based techniques are also expected to do well in these scenarios. However, they normally return many variants of the ideal clones and that dissimilar code fragments can lead to similar graphs leading to low precision.

6.3. Scenario 3

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 2).*

We hope that an ideal clone detection technique would detect all five fragments as clone pairs with the original and form a clone class for them. Again, code fragments of *Scenario 1* and *Scenario 2* might also form clone pairs / classes with the code fragments of this scenario. The fifth column under the *Scenario 3* heading of Table 14 summarizes how well each technique may work on these scenarios.

In general, text-based techniques and tools are not good at detecting *Type-3* near-miss clones created using *Scenario 3* unless they apply threshold-based comparison or combine smaller *Type-1* and *Type-2* clones in a post-processing phase. *Duploc* transforms program text to a condensed form (removing whitespace and comments) then applies string-based dynamic pattern matching with gaps, and hence can detect changes within a line. Therefore, *Duploc* is expected to do well on scenarios S3(a) and S3(b) (and possibly S3(e)). Although *DuDe* [114] is text-based, it can combine small duplicated segments to form larger ones by allowing gaps in its scatter plot visualization. Both *Basic* NICAD [99] and *Full* NICAD [104] detect these scenarios well as they allow size-sensitive threshold-based comparison of the extracted and pretty-printed potential clones. *Full* NICAD [104] also uses flexible code normalization and filtering that removes many of the small differences between code fragments and thus can also detect *Type-3* clones. Nasehi's approach [92] transforms code to semantically equivalent atomic units and uses an edit distance algorithms

with allowable thresholds. Thus, this approach is also expected to detect scenarios S3(a) and S3(b) well, and possibly also scenarios S3(c), S3(d), and S3(e).

Among the token-based techniques, only *Gemini* [112] (a post-processor / visualizer for *CCFinder* [59]) and *CP-Miner* [84] are likely to work well with these scenarios. *CP-Miner* uses a frequent subsequence data mining algorithm which allows it to tolerate gaps in cloned segments. *Gemini* on the other hand, uses output (*Type-1* and *Type-2* clones) from *CCFinder* and scatter plot visualization to detect such near-miss clones, much like *DuDe*.

Among the tree-based techniques, only *Deckard* [52] and *Asta* [42] are likely to do well for these scenarios. *Asta* derives syntax-tree patterns with placeholders for complete subtrees, which supports structural abstraction. *Deckard* uses the novel idea of a characteristics vector (thus *Deckard* can also be classified as a metrics-based tool) to approximate the structural information from ASTs in the Euclidean space. However, as with metrics-based approaches, such an approximation is challenging and vector values from two quite distinct code fragments may be similar, indicating that *Deckard* could return many false positives in detecting such clones. Other tree-based tools, such as *CloneDr* [15] and *ccdiml* [14], may detect scenarios S3(a) and S3(b) if their underlying similarity measure for inexact tree matching is set to tolerate them. In *CloneDr*, a compiler generator is used to generate an annotated parse tree (AST) and compares its subtrees by characterization metrics based on a hash function. Source code of similar subtrees is then returned as clones. The hash function enables one to do parameterized matching and to detect gapped clones, especially if the gaps are within a line. *ccdiml* is a variant of *CloneDR* that has a different intermediate representation with explicit modeling of sequences, which helps in finding near-miss clones created from these scenarios.

Metrics-based techniques can find clones in these scenarios, but may yield many false positives, since many other code fragments may have similar metrics values, resulting in lower overall accuracy. However, scenarios S3(a) and S3(b) can likely be accurately detected by at least some of the metrics-based approaches, notably Mayrand's [87], Dagenais's [30], Merlo's [90] and Antoniol's [2].

Graph-based approaches primarily use control and data flow information and thus are expected to detect these scenarios well. In fact, in Bellon's experiment [18], the graph-based tool *Duplix* [75] was found to detect a small proportion of such near-miss clones. However, in general graph-based tools may return many variants of the ideal clones, and some of these variants can be considered as false positives, yielding a lower overall accuracy. Only the recent semantics-based approach by Gabel et al. [46] has been demonstrated to scale. Instead of comparing subgraphs of the PDGs, Gabel's approach maps subgraphs to related structured syntax and then finds clones using *Deckard* technique.

One should also note that although Kamiya [59], Krinke [75], Mayrand/Merlo [90, 87], and Rieger [41] mention that their approaches can also find clones of *Type-3*, according to Bellon et al.'s study [18] in practice only Krinke's *Duplix* actually does. In *Duplix*, however, clones of other types are found with very low recall.

6.4. Scenario 4

Scenario 4: *The programmer makes four more copies of the function and this time reorders the data independent declarations 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 a different one in the fourth (S4(d)) (Figure 2).

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. Once again, code fragments of *Scenario 1*, *Scenario 2* and *Scenario 3* might form clone pairs/clone classes with the code fragments of this scenario. The sixth column under the *Scenario 4* heading of Table 14 summarizes how well each technique is likely to work in these scenarios.

Among the text-based techniques, only Marcus's LSI approach [86] is likely to do well with scenarios S4(a), S4(b) and S4(c). Marcus's approach considers only comments and identifier names in the comparison. When statements of copied fragments are reordered, comments and identifiers may not be changed and thus their approach may detect these scenarios. Nasehi's approach [92] performs a semantics-preserving transformation for different syntactic variants of a language to the same atomic units. Thus, the representation of the atomic units of the original function with the *for loop* might be similar to the atomic representation of the copied function with *while loop* of scenario S4(d). Moreover, this approach uses an edit distance based algorithm, which allows for dissimilarity thresholds in the comparison. We therefore expect that Nasehi's approach may be able to detect clones created by scenario S4(d). NICAD probably can detect the reordering scenarios S4(a), S4(b) and S4(c) if the total gap created by the reordering of statements is within the allowable size-sensitive difference thresholds. However, increasing the threshold might lead to false positive clones.

Unfortunately, there is no token-based technique that can detect clones created in these scenarios well. This is obvious since these techniques/tools use exact matching on normalized token sequences and do not allow for any gaps. Reordering statements (scenarios S4(a), S4(b) and S4(c)) or replacements of one control by another equivalent variant (scenario S4(d)) obviously breaks the token sequences between the original and copied code fragments. However, some token-based tools, such as *Gemini* and *CP-Miner*, might detect scenarios S4(a), S4(b) and S4(c). *Gemini* uses scatter plot visualization of *Type-1* and *Type-2* clones from *CCFinder* and thus might detect scenarios S4(a), S4(b) and S4(c) by allowing gaps. *CP-Miner* allows for arbitrary gaps in cloned segments and thus might also detect scenarios S4(a), S4(b) and S4(c). However, there is no token-based technique that can detect scenario S4(d).

The situation is worse in the case of tree-based techniques. There is no tree-based technique or tool that can be expected to detect these scenarios, with the possible exception that *CloneDr* may be able to detect clones of scenario S4(a) since its subtree characterization can ignore declaration statements.

Metrics-based techniques should be able to detect scenarios S4(a) and S4(b) well, since reordering of data-independent statements might not change the metrics values. However, metrics values might change when reordering happens between data-dependent statements (scenario S4(c)) due to the underlying metrics definition. When control replacement is performed on the copied fragment (scenario S4(d)) metrics values might change significantly and thus metrics-based techniques either cannot detect scenario S4(d) or will detect it with many false positives, yielding a low overall accuracy.

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 the data and control flow graphs however, so they may not do as well with scenario S4(c). To detect scenario S4(d), exhaustive source transformation may be necessary. However, an alternative approach is proposed in the plagiarism detection tool *GPLAG* [81] for finding plagiarized code similar to those created by scenario S4(d).

7. An Example Use of the Study

Our survey and evaluations are not just intended for experts in clone detection, but also for users and builders of tools based on clone detection techniques. As a demonstration of how this study can help, we provide two example user intentions and suggest a tool or set of tools to meet their requirements. Of course, many other combinations of the tools 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 approach to be robust across all types of clones or how to employ a set of different tools to achieve a better result. Our NICAD tool [104] is an example of such a hybrid, combining tree-based structural analysis with text-based comparison.

Intention 1: *A tool user would like to find all types of clones (as outlined in this paper) in a large C system (the Linux kernel) with reasonable performance.*

Here, the primary objective is the ability to handle large C systems while doing well in finding all the kinds of clones that may be created by the various editing scenarios outlined in Section 6. Let us first look for individual tools that rate reasonably well for the scenarios. From Table 14 we see that the obvious set is *Gabel [46], GPLAG [81], Kontogiannis [66], C2D2 [74], CP-Miner [84], Gemini [112], Nasehi [92].* Although none of these tools is able to handle all of the scenarios, they all seem to do well with most of the scenarios.

The second requirement is that the tool should handle C systems. According to our findings in Section 5 (column 5 of Table 12), only *Gabel* [46], *GPLAG* [81], Kontogiannis [66], *CP-Miner* [84], and *Gemini* [112] meet this requirement.

As a third requirement, the user needs a tool capable of handling large systems. From the 7th column with column heading *Technical Aspects* and from the corresponding facet table (Table 6), the user can get an idea of the algorithms used and their associated complexities. In particular from the sub-column *Computational Complexity* of Table 12 and from the last row (*CC (Worst Case Computational Complexity)* of Table 6, we see that of the set we have chosen only *Gemini* seems computationally efficient. However, although *Gemini* uses *CCFinder* in the background for finding smaller *Type-1* and *Type-2* clones which is linear w.r.t. the size of the program, finding combination of *Type-1* and *Type-2* clones to form *Type-3* clones may require superlinear time; often dynamic programming is used for this combination, which is not linear. Furthermore, *Gemini* is mainly a visualization tool and thus might not fully meet our user's requirements.

The question now remains as to whether there are other tools in our candidate set that can handle large systems despite having non-linear (worst case) computational complexities. We can find this information from the 10th column (with column heading *Evaluation*) of Table 12. In particular, from the sub-column with heading *Subject Systems* and the corresponding *Evaluation* facet table (Table 11) we see that fortunately both *CP-Miner* and *Gabel* have been evaluated even with *Linux Kernel*, one of the largest C systems. The question again remains whether the results

of their studies are available, especially for the *Linux Kernel*. We can see this information in the same column *Evaluation* with sub-column *Availability of Results* and the corresponding facet table (with row heading *AR (Availability of Results)* in Table 11). We see that complete results are not available for *Linux Kernel*. Moreover, *Linux* is changing every day and thus results for the intended version might not be available anyway.

As the results are not available, the user needs to run the tool (either *CP-Miner* or *Gabel*) to find clones in *Linux*. The next crucial question now is whether the tools are available for third party use. From the *Availability* facet of *Usage* category in Table 12 and in more detail in Table 2, we observe that neither of them is available online but an evaluation version may be available on request.

The remaining question is, which tool to request first? The user can ask for both tools, or can be more specific in determining who might actually be able to release their tool. In particular, we can look to see whether the tool is standalone or has any external dependencies or is a part of required larger tool set. If the tool is standalone, it is more likely that the tool will be available upon request, otherwise it is likely that the tool may not be available, or even if available may be hard to use by a third party. Unfortunately, we see (from the same tables above) that both the tools have external dependencies. With a closer look in the description of the row *External Dependencies* in Table 2 we see that *CP-Miner* is dependent on *CloSpan* and *Gabel* is dependent on *CodeSurfer*. Given that both tools are dependent on other systems, the user might contact both the tool authors or may choose to undertake further studies on the dependencies by reading the details in Section 4 or the corresponding papers before contacting the tool authors. The user might also reconsider other tools because neither of the chosen two can actually detect all types of clones. Using our study and evaluations in this paper, one can identify options quickly and with minimal effort.

Intention 2: *A user wants to detect clones from many systems in different languages. The user does not aim to detect all types of clones but the intention is to detect as many types as possible. The user is also willing to do some adaptation work for different languages if the tool is really good. Computational complexity should be reasonable but need not be ideal.*

Here, the primary concern of the user is that the tool should be either language-independent or adaptable to other languages with a reasonable amount of effort. However, there is a trade-off between the adaptability to different languages and the quality (e.g., cabability of dealing with different types of clones) of the tool. The user is also comfortable with computationally expensive tools, within reason (i.e., not taking weeks or months to process systems).

From Table 10 we find that text-based tools are either language-independent or easily adaptable to other languages but the computational complexity depends on the algorithm used. Token-based tools are in most cases language-dependent (needing a lexer at least) but computationally efficient. As the concern is adaptability (and not the complexity), the user looks for a text-based tool (or set of tools) in Table 14 that covers most of the scenarios. Such a set of tools is *Full* NICAD [104] and *Nasehi* [92]. Before taking the final decision of which tool should be chosen, the user needs to look at some details of the tools. In particular, the user needs to know whether the chosen tools have any language dependencies or not, and in case there is any language dependency, how much effort it might take to adapt to other languages.

From Section 5 we know that the *Program Analysis* facet under category *Processing* has information regarding language dependencies. Two other facets (*Transformation* and *Code Represen-*

tation) of category *Processing* further hint about the adaptability of a tool to a different language. From the sub-column *Program Analysis* of the 9th column of Table 12 we see that both tools have attribute value g for the *Program Analysis* facet. Table 10 tells us that this means that both of the tools use language-dependent transformation rules. Thus even though they are text-based techniques, they might not be easily adaptable to other languages since they apply advanced transformation rules on the program text before the comparison. These transformations obviously need syntactic (or semantic) analysis of the source code.

In order to gain further insight into the tools, we examine the attribute values of the other two facets, *Transformation* and *Code Representation*, and find that while NICAD uses example-like code normalization rules, which may easily port to other languages, *Nasehi* applies semantics-preserving transformations to yield an equivalent set of atomic instructions, which may not be as easy to adjust. Thus NICAD may require less work to adapt than *Nasehi*.

In this situation the user may choose NICAD or may compare other attributes of the two tools to come to a final decision. In particular, the user can examine the *Language* facets (*Language Paradigm* and *Language Support*) of the two tools. From the 5th column of Table 12 (and the associated facet Table 4) we see that while NICAD can handle both procedural (e.g., C) and object-oriented (e.g., Java) systems, *Nasehi* works only with object-oriented systems (Java) and thus, *Full* NICAD may be a better choice than *Nasehi* for this purpose. Of course, other facet attributes should also be examined for a final decision.

Alternatively, the user may look for token-based tools. At a first glance at Table 14, we see that *CP-Miner* [84] covers most of the clone types/sub-scenarios. However, after examining its attribute values from Table 12 (and the associated facet tables) we find that a full-fledged parser is required when it needs to be adapted to a different language, and that it depends on an external system (*CloSpan*). The user thus cannot choose *CP-Miner*.

Instead of giving up on which tool to choose (there are about 40 tools out there), the user can examine the *Program Analysis* facet table (Table 10) first. This table shows the different attributes (with description) of language dependency and citations to the corresponding tools. Fortunately, we see that attribute *PA.h: Needs only a context-free grammar for the language dialect of interest* has three citations, one text-based (denoted with T) tool, *Basic* NICAD [99], one token-based (denoted with L), *Cordy* [28], and one tree-based (denoted with S) tool, *Deckard* [52]. Among the three tools, *Cordy* only works with HTML pages and thus cannot be chosen as the user wants to find clones in different procedural and object-oriented systems.

The question now remains whether to choose the text-based *Basic* NICAD or the tree-based *Deckard*. The user then has to examine which tool covers most of the clone types. From Table 14 we see that *Deckard* covers more clone types/sub-scenarios than *Basic* NICAD. Furthermore, even though a tree-based tool, *Deckard* needs only a context-free grammar to adapt to a new language. *Basic* NICAD also only needs a context-free grammar, but has to be written in TXL [27] format. Of course, the user has to examine the other associated attribute values of the two tools before coming to a conclusion.

These two examples demonstrate some of the ways how our study can be used to assist in understanding the alternatives when faced with a need for clone detection. Depending on the particular intentions, a range of possibilities may present themselves, but using our summary tables, alternatives can be quickly narrowed down to focus on the one or two most appropriate to the

application.

8. Related Work

Although there is no work in the literature that provides a property-based comparison and scenario-based evaluation of the techniques and tools similar to this study, several tool comparison experiments have been conducted to estimate the abilites of the tools in terms of precision, recall, and time and space requirements.

One of the first experiments was conducted by Bailey and Burd [22], who compared three state-of-the-art clone detection and two plagiarism detection tools. They began by validating all the clone candidates of the subject application obtained with all the techniques of their experiment to form a human oracle, which was then used to compare the different techniques in terms of several metrics to measure various aspects of the reported clones.

Although they were able to verify all the clone candidates, the limitations of the case study in terms of a single subject system, modest system size and validation subjectivity may make their findings less than definitive. Moreover, the intention of their analysis was to assist in preventative maintenance tasks, which may have influenced their clone validation process.

Considering the limitations of Burd and Bailey's study, Bellon et al. set out to conduct a larger tool comparison experiment [18] on the same three clone detection tools used in Burd and Bailey's study and three additional clone detection tools. They also used a more diverse set of larger software systems, consisting of four Java and four C systems totaling almost 850 KLOC. As in the study of Burd and Bailey, a human oracle validated a random sample of about 2% of the candidate clones from all the tools evenly and blindly. While their study is the most extensive to date, only a small proportion of the clone candidates were oracled and several other factors may have influenced the results [4]. Bellon's framework has been reused in experiments by Koschke et al. [72, 43] and Ducasse et al. [40] (partially), but without any improvements to the framework.

Rysselberghe and Demeyer [106, 105] have evaluated prototypes of three representative clone detection techniques, providing comparative results in terms of portability, kinds of duplication reported, scalability, number of false matches, and number of useless matches. However, they did not make a reference set, used relatively small subject systems (under 10 KLOC) and did not provide the reliability of the judge(s) that validated the found clones. Moreover, rather than quantitative evaluation of the detection techniques, their intention was to determine the suitability of the clone detection techniques for a particular maintenance task (refactoring) which might have influenced their clone validation.

Another interesting study has been conducted by Bruntink et al.[19], in which several clone detection techniques are evaluated in terms of finding known cross-cutting concerns in C programs with homogeneous implementations.

9. Conclusion

In this paper, we have focused on clone detection techniques and tools, providing a concise but comprehensive survey and a hypothetical evaluation based on editing scenarios. A more detailed review of the entire range of clone detection research can be found in our technical report [102].

Koschke's Dagstuhl report [73] and the corresponding book chapter [70] also provide 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 techniques and tools 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. [18].

While in this study our goal was predictive rather than empirical, we are currently undertaking an experiment using our editing scenarios as the basis for generating and injecting thousands of artificial mutants which can be used to empirically compare actual tools on a similar basis [98, 101].

Acknowledgements: We thank the anonymous reviewers for their valuable comments and suggestions in improving this 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 and by an IBM international faculty award.

References

[1] G. Antoniol, G. Casazza, M. Di Penta, E. Merlo, Modeling Clones Evolution through Time Series, in: Proceedings of the 17th IEEE International Conference on Software Maintenance, ICSM 2001, pp. 273-280 (2001).

[2] G. Antoniol, U. Villano, E. Merlo, and M.D. Penta, Analyzing cloning evolution in the linux kernel, Information and Software Technology, 44 (13):755-765 (2002).

[3] L. Aversano, L. Cerulo, and M. Di Penta, How Clones are Maintained: An Empirical Study, in: Proceedings of the 11th European Conference on Software Maintenance and Reengineering, CSMR 2007, pp. 81-90 (2007).

[4] B. Baker, Finding Clones with Dup: Analysis of an Experiment, IEEE Transactions on Software Engineering, 33(9):608-621, (2007).

[5] B. Baker and U. Manber, Deducing similarities in Java sources from bytecodes, in: Proceedings of the USENIX Annual Technical Conference, pp. 179-190, (1998).

[6] B. Baker, A Program for Identifying Duplicated Code, in: Proceedings of Computing Science and Statistics: 24th Symposium on the Interface, Vol. 24:4957, 24:49-57 (1992).

[7] B. Baker, Parameterized Pattern Matching: Algorithms and Applications, Journal Computer System Science, Vol. 52(1):28-42 (1996).

[8] B. Baker, On Finding Duplication and Near-Duplication in Large Software Systems, in: Proceedings of the 2nd Working Conference on Reverse Engineering, WCRE 1995, pp. 86-95 (1995).

[9] B. Baker and R. Giancarlo, Sparse Dynamic Programming for Longest Common Subsequence from Fragments, Journal Algorithms, Vol. 42 (2):231-254 (2002).

[10] M. Balazinska, E. Merlo, M. Dagenais, B. Lague and K. Kontogiannis, Measuring Clone Based Reengineering Opportunities, in: Proceedings of the IEEE Symposium on Software Metrics, METRICS 1999, pp. 292-303 (1999).

[11] M. Balint, T. Girba, R. Marinescu, How Developers Copy, in: Proceedings of the 14th IEEE International Conference on Program Comprehension, ICPC 2006, pp. 56-68 (2006).

[12] H. Basit, S. Pugliesi, W. Smyth, A. Turpin and S.Jarzabek, Efficient Token Based Clone Detection with Flexible Tokenization, in: Proceedings of the 6th European Software Engineering Conference and Foundations of Software Engineering, ESEC/FSE 2007, pp. 513-515 (2007).

[13] I. D. Baxter, C. Pidgeon, and M. Mehlich, DMS: Program Transformations for Practical Scalable Software Evolution, in: Proceedings of the 26th International Conference on Software Engineering, ICSE 2004, pp. 625-634 (2004).
[14] Project Bauhaus. URL `http://www.bauhaus-stuttgart.de` Last accessed November 2008.
[15] I. Baxter, A. Yahin, L. Moura and M. Anna, Clone Detection Using Abstract Syntax Trees, in: Proceedings of the 14th International Conference on Software Maintenance, ICSM 1998, pp. 368-377 (1998).
[16] S. Bellon, Vergleich von techniken zur erkennung duplizierten quellcodes, Diploma Thesis, University of Stuttgart, 2002.
[17] S. Bellon and R. Koschke, Detection of Software Clone: Tool Comparison Experiment, `http://www.bauhaus-stuttgart.de/clones/` (December 2007).
[18] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo, Comparison and Evaluation of Clone Detection Tools, Transactions on Software Engineering, 33(9):577-591 (2007).
[19] M. Bruntink, A. Deursen, R. Engelen and T. Tourwe, On the Use of Clone Detection for Identifying Crosscutting Concern Code, Transactions on Software Engineering, 31(10):804-818 (2005).
[20] P. Bulychev and M. Minea, Duplicate Code Detection Using Anti-Unification, in: Spring Young Researchers Colloquium on Software Engineering, SYRCoSE 2008, 4 pp. (2008).
[21] P. Bulychev, CloneDigger Results: http://clonedigger.sourceforge.net/ Last accessed February 2009.
[22] E. Burd, J. Bailey, Evaluating Clone Detection Tools for Use during Preventative Maintenance, in: Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation, SCAM 2002, pp. 36-43 (2002).
[23] F. Calefato, F. Lanubile and T. Mallardo, Function Clone Detection in Web Applications: A Semiautomated Approach, Journal of Web Engineering, 3(1):3-21 (2004).
[24] G. Casazza, G. Antoniol, U. Villano, E. Merlo and M. Penta, Identifying Clones in the Linux Kernel, in: Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation, SCAM 2001, pp. 90-97, (2001).
[25] K. Church and J. Helfman, Dotplot: A Program for Exploring Self-similarity in Millions of Lines for Text and Code, Journal of American Stat. Ass., 2(2):153-174 (1993).
[26] K. Cooper and N. McIntosh, Enhanced code compression for embedded risc processors, in: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation, SIGPLAN PLDI 1999, pp. 139-149 (1999).
[27] J.R. Cordy, The TXL Source Transformation Language, Science of Computer Programming, 61(3):190-210 (2006).
[28] J.R. Cordy, T.R. Dean and N. Synytskyy, Practical Language-Independent Detection of Near-Miss Clones, in: Proceedings of the 14th IBM Centre for Advanced Studies Conference, CASCON 2004, pp. 29-40 (2004).
[29] PMD's CPD. URL `http://pmd.sourceforge.net/cpd.html` Last accessed November 2008.
[30] M. Dagenais, E. Merlo and B. Laguë, and Daniel Proulx, Clones Occurrence in Large Object Oriented Software Packages, in: Proceedings of the 8th IBM Centre for Advanced Studies Conference, CASCON 1998, pp. 192-200 (1998).
[31] N. Davey, P. Barson, S. Field and R. Frank, The Development of a Software Clone Detector, International Journal of Applied Software Technology, 1(3/4):219-236 (1995).
[32] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, Compiler Techniques for Code Compaction, ACM Transactions on Programming Languages and Systems, Vol. 22(2):378-415 (2000).
[33] F. Deissenboeck, B. Hummel, E. Juergens, B. Schaetz, S. Wagner, S. Teuchert and J. Girard, Clone Detection in Automotive Model-Based Development, in: Proceedings of the 30th International Conference on Software Engineering, ICSE 2008, pp. 603-612 (2008).
[34] B. De Sutter, B. De Bus, and K. De Bosschere, Sifting out the mud: Low level C++ code reuse, in: Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA 2002, pp. 275-291 (2002).
[35] A. De Lucia, R. Francese, G. Scanniello, and G. Tortora, Understanding Cloned Patterns in Web Applications, in: Proceedings of the 13th International Workshop on Program Comprehension, IWPC 2005, pp. 333-336 (2005).

[36] G.A. Di Lucca, M. Di Penta, A.R. Fasolino and P. Granato, Clone Analysis in the Web Era: an Approach to Identify Cloned Web Pages, in: Proceedings of the 7th IEEE Workshop on Empirical Studies of Software Maintenance, WESS 2009, pp. 107-113 (2001).

[37] Tool Clone Detective (part of ConQAT). URL http://conqat.in.tum.de/index.php/Main_Page Last accessed November 2008.

[38] G. Di Lucca, M. Penta and A. Fasolino, An Approach to Identify Duplicated Web Pages, in: Proceedings of the 26th International Computer Software and Applications Conference, COMPSAC 2002, pp. 481-486 (2002).

[39] Tool Dupman. URL `http://sourceforge.net/projects/dupman` Last accessed November 2008.

[40] S. Ducasse, O. Nierstrasz and M. Rieger, On the Effectiveness of Clone Detection by String Matching, International Journal on Software Maintenance and Evolution: Research and Practice, 18(1): 37-58 (2006).

[41] S. Ducasse, M. Rieger and S. Demeyer, A Language Independent Approach for Detecting Duplicated Code, in: Proceedings of the 15th International Conference on Software Maintenance, ICSM 1999, pp. 109-118 (1999).

[42] W. Evans, C. Fraser and M. Fei, Clone Detection via Structural Abstraction, in: Proceedings of the 14th Working Conference on Reverse Engineering, WCRE 2007, pp. 150-159 (2007).

[43] R. Falke, R. Koschke and P. Frenzel, Empirical Evaluation of Clone Detection Using Syntax Suffix Trees, Empirical Software Engineering, Vol. 13: 601-643 (2008).

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

[45] C. Fraser, E. Myers, and A. Wendt, Analyzing and compressing assembly code, in: Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction, pp. 117-121 (1984).

[46] M. Gabel, L. Jiang and Z. Su, Scalable Detection of Semantic Clones, in: Proceedings of the 30th International Conference on Software Engineering, ICSE 2008, pp. 321-330 (2008).

[47] D. Gitchell and N. Tran, Sim: A Utility for Detecting Similarity in Computer Programs, SIGCSE Bulletin, 31(1): 266-270 (1999).

[48] N. Göde, Incremental Clone Detection, Diploma Thesis, Department of Mathematics and Computer Science, University of Bremen, Germany, 2008.

[49] J. Guo and Y. Zou, Detecting Clones in Business Applications, in: Proceedings of the 15th Working Conference on Reverse Engineering, WCRE 2008, pp. 91-100 (2008).

[50] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto and K. Inoue, On Software Maintenance Process Improvement Based on Code Clone Analysis, in: Proceedings of the 4th International Conference on Product Focused Software Process Improvement, PROFES 2002, pp. 185-197 (2002).

[51] P. Jablonski and D. Hou, CReN: A Tool for Tracking Copy-and-paste Code Clones and Renaming Identifiers Consistently in the IDE, in: Proceedings of Eclipse Technology Exchange Workshop at OOPSLA 2007, pp. 16-20 (2007).

[52] L. Jiang, G. Misherghi, Z. Su and S. Glondu, DECKARD: Scalable and Accurate Tree-based Detection of Code Clones, in: Proceedings of the 29th International Conference on Software Engineering, ICSE 2007, pp. 96-105 (2007).

[53] J. Johnson, Identifying Redundancy in Source Code Using Fingerprints, in: Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON 1993, pp. 171–183 (1993).

[54] J. Johnson, Visualizing Textual Redundancy in Legacy Source, in: Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative research, CASCON 2004, pp. 171-183 (1994).

[55] J. Johnson, Substring Matching for Clone Detection and Change Tracking, in: Proceedings of the 10th International Conference on Software Maintenance, ICSM 1994, pp.120-126 (1994).

[56] E. Juergens, F. Deissenboeck, B. Hummel and S. Wagner, Do Code Clones Matter?, in: Proceedings of the 31st International Conference on Software Engineering, ICSE 2009, 11 pp. (2009)(to appear).

[57] N. Juillerat, and B. Hirsbrunner, An Algorithm for Detecting and Removing Clones in Java Code, in: Proceedings of the 3rd Workshop on Software Evolution through Transformations: Embracing the Change, SeTra 2006, pp.63-74 (2006).

[58] T. Kamiya, The Official CCFinderX Website. URL `http://www.ccfinder.net/ccfinderx.html` Last accessed November 2008.

[59] T. Kamiya, S. Kusumoto and K. Inoue, CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code, IEEE Transactions on Software Engineering, 28(7):654-670 (2002).

[60] C. Kapser, and M. Godfrey, Aiding Comprehension of Cloning Through Categorization, in: Proceedings of the 7th International Workshop on Principles of Software Evolution, IWPSE 2004, pp. 85-94 (2004).
[61] Cory Kapser and Michael W. Godfrey, "Cloning Considered Harmful" Considered Harmful: Patterns of Cloning in Software, Empirical Software Engineering, Vol. 13(6):645-692 (2008).
[62] H. M. Kienle, H. A. Müller, and A. Weber, In the Web of Generated "Clones", in: Proceedings of 2nd International Workshop on Detection of Software Clones, IWDSC 2003, 2pp. (2003).
[63] M. Kim, L. Bergman, T. Lau and D. Notkin, An Ethnographic Study of Copy and Paste Programming Practices in OOPL, in: Proceedings of 3rd International ACM-IEEE Symposium on Empirical Software Engineering, ISESE 2004, pp. 83–92 (2004).
[64] M. Kim, G. Murphy, An Empirical Study of Code Clone Genealogies, in: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/SIGSOFT FSE 2005, pp. 187-196 (2005).
[65] R. Komondoor and S. Horwitz, Using Slicing to Identify Duplication in Source Code, in: Proceedings of the 8th International Symposium on Static Analysis, SAS 2001, pp. 40-56 (2001).
[66] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein, Pattern Matching for Clone and Concept Detection, Journal of Automated Software Engineering, 3(1-2):77-108 (1996).
[67] K. Kontogiannis, Evaluation Experiments on the Detection of Programming Patterns using Software Metrics, in: Proceedings of the 3rd Working Conference on Reverse Engineering, WCRE 1997, pp. 44-54 (1997).
[68] S. Rao Kosaraju, Faster algorithms for the construction of parameterized suffix trees, in: Proceedings of the 36th Annual Symposium on Foundations of Computer Science, FOCS 1995, pp. 631-638 (1995).
[69] R. Koschke, Frontiers on Software Clone Management, in: Proceedings of the Frontiers of Software Maintenance, in: 24th IEEE International Conference in Software Maintenance, ICSM 2008, pp. 119-128 (2008).
[70] R. Koschke, Identifying and Removing Software Clones, in: Software Evolution, pp. 15-39, Springer Verlag, Editors: Serge De- meyer und Tom Mens (2008).
[71] R. Koschke, J.-F. Girard, M. Wrthner, An Intermediate Representation for Reverse Engineering Analyzes, in: Proceedings of the 5th Working Conference on Reverse Engineering, WCRE 1998, pp. 241-250 (1998).
[72] R. Koschke, R. Falke and P. Frenzel, Clone Detection Using Abstract Syntax Suffix Trees, in: Proceedings of the 13th Working Conference on Reverse Engineering, WCRE 2006, pp. 253-262 (2006).
[73] R. Koschke, Survey of Research on Software Clones, in: Proceedings of Dagstuhl Seminar 06301: Duplication, Redundancy, and Similarity in Software, 24pp. (2006).
[74] N. Kraft, B. Bonds and R. Smith, Cross-Language Clone Detection, in: Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering, SEKE 2008, 6 pp. (2008).
[75] J. Krinke, Identifying Similar Code with Program Dependence Graphs, in: Proceedings of the 8th Working Conference on Reverse Engineering, WCRE 2001, pp. 301-309 (2001).
[76] I. Landwerth, Clone Detective, URL `http://www.codeplex.com/CloneDetectiveVS` Last accessed November 2008.
[77] F. Lanubile and T. Mallardo, Finding Function Clones in Web Applications, in: Proceedings of the 7th European Conference on Software Maintenance and Reengineering, CSMR 2003, pp. 379–386 (2003).
[78] S. Lee and I. Jeong, SDD: High performance Code Clone Detection System for Large Scale Source Code, in: Proceedings of the Object Oriented Programming Systems Languages and Applications Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA Companion 2005, pp. 140–141 (2005).
[79] A. Leitão, Detection of Redundant Code Using R^2D^2, Software Quality Journal, 12(4):361-382 (2004).
[80] H. Li and S. Thompson, Clone Detection and Removal for Erlang/OTP within a Refactoring Environment, in: ACM/SIGPLAN Workshop Partial Evaluation and Semantics-Based Program Manipulation, in: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation, pp. 169–178 (2009).
[81] C. Liu, C. Chen, J. Han and P. Yu, GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis, in: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2006, pp. 872-881 (2006).
[82] H. Liu, Z Ma, L. Zhang and W Shao, Detecting Duplications in Sequence Diagrams Based on Suffix Trees, in: Proceedings of the 13th Asia Pacific Software Engineering Conference, APSEC 2006, pp. 269-276 (2006).

[83] S. Livieri, Y. Higo, M. Matsushita and K. Inoue, Very-Large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder, in: Proceedings of 29th International Conference on Software Engineering, ICSE 2007, pp. 106-115 (2007).

[84] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code, IEEE Transactions on Software Engineering, 32(3):176-192 (2006).

[85] U. Manber, Finding Similar Files in a Large File System, in: Proceedings of the Winter 1994 Usenix Technical Conference, pp. 1-10 (1994).

[86] A. Marcus and J. Maletic, Identification of High-level Concept Clones in Source Code, in: Proceedings of the 16th IEEE International Conference on Automated Software Engineering, ASE 2001, pp. 107-114 (2001).

[87] J. Mayrand, C. Leblanc and E. Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics, in: Proceedings of the 12th International Conference on Software Maintenance, ICSM 1996, pp. 244-253 (1996).

[88] E. McCreight, A Space-economical Suffix Tree Construction Algorithm, Journal of the ACM, 32(2):262-272 (1976).

[89] E. Merlo, M. Dagenais, P. Bachand, J.S. Sormani, S. Gradara and G. Antoniol, Investigating Large Software System Evolution: the Linux Kernel, in: Proceedings of the 26th International Computer Software and Applications Conference, COMPSAC 2002, pp. 421-426 (2002).

[90] E. Merlo, G. Antoniol, M. Penta and V. Rollo, Linear Complexity Object-Oriented Similarity for Clone Detection and Software Evolution Analyses, in: Proceedings of the 20th International Conference Conference on Software Maintenance, ICSM 2004, pp. 412–416 (2004).

[91] L. Moonen, Generating Robust Parsers Using Island Grammars, in: Proceedings of the 8th Working Conference on Reverse Engineering, WCRE 2001, pp. 13-22 (2001).

[92] S. Nasehi, G. Sotudeh and M. Gomrokchi, Source Code Enhancement Using Reduction of Duplicated Code, in: Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering, IASTED SE 2007, pp. 192-197 (2007).

[93] J. Patenaude, E. Merlo, M. Dagenais and B. Lague, Extending Software Quality Assessment Techniques to Java Systems, in: Proceedings of the 7th International Workshop on Program Comprehension, IWPC 1999, pp. 49–56 (1999).

[94] D. Rajapakse, and S. Jarzabek, Using Server Pages to Unify Clones in Web Applications: A Trade-off Analysis, in: Proceedings of the 29th International Conference of Software Engineering, ICSE 2007, pp. 116–126 (2007).

[95] F. Ricca and P. Tonella, Using Clustering to Support the Migration from Static to Dynamic Web Pages, in: Proceedings of the 11th International Workshop on Program Comprehension, IWPC 2003, pp. 207-216 (2003).

[96] M. Rieger. Effective Clone Detection Without Language Barriers, Ph.D. Thesis, University of Bern, Switzerland, 2005.

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

[98] C.K. Roy and J.R. Cordy, A Mutation / Injection-based Automatic Framework for Evaluating Clone Detection Tools, in: Proceedings of the 4th International Workshop on Mutation Analysis, Mutation 2009, 10pp. (2009)(submitted).

[99] C.K. Roy and J.R. Cordy, An Empirical Study of Function Clones in Open Source Software Systems, in: Proceedings of the 15th Working Conference on Reverse Engineering, WCRE 2008, pp. 81-90 (2008).

[100] C.K.Roy and J.R. Cordy, WCRE'08 Clones, http://www.cs.queensu.ca/home/stl/download/NICADOutput/ Last accessed November 2008.

[101] C.K. Roy and J.R. Cordy, Towards a Mutation-Based Automatic Framework for Evaluating Clone Detection-Tools, in: Proceedings of the Canadian Conference on Computer Science and Software Engineering, C3S2E 2008, pp. 137-140 (2008).

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

[103] C.K. Roy and J.R. Cordy, Scenario-Based Comparison of Clone Detection Techniques, in: Proceedings of the 16th IEEE International Conference on Program Comprehension, ICPC 2008, pp. 153-162 (2008).

[104] C.K. Roy and J.R. Cordy, NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization, in: Proceedings of the 16th IEEE International Conference on Program Comprehension, ICPC 2008, pp. 172-181 (2008).

[105] F. Rysselberghe and S. Demeyer, Evaluating Clone Detection Techniques, in: Proceedings of the International Workshop on Evolution of Large Scale Industrial Applications, ELISA 2003, 12pp. (2003).

[106] F. Rysselberghe and S. Demeyer, Evaluating Clone Detection Techniques from a Refactoring Perspective, in: Proceedings of the 9th IEEE International Conference Automated Software Engineering, ASE 2004, pp. 336-339 (2004).

[107] Tool Simian, URL `http://www.redhillconsulting.com.au/products/simian/` Last accessed November 2008.

[108] Tool SimScan, URL `http://www.blue-edge.bg/simscan/` Last accessed November 2008.

[109] T. Sager, A. Bernstein, M. Pinzger and C. Keifer, Detecting Similar Java Classes Using Tree Algorithms, in: Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, pp. 65-71 (2006).

[110] N. Synytskyy, J. R. Cordy and T. R. Dean, Resolution of Static Clones in Dynamic Web Page, in: Proceedings of the 5th IEEE International Workshop on Web Site Evolution, WSE 2003, pp. 49-58 (2003).

[111] R. Tairas and J. Gray, Phoenix-Based Clone Detection Using Suffix Trees, in: Proceedings of the 44th Annual Southeast Regional Conference, ACM-SE 2006, pp. 679-684 (2006).

[112] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, On Detection of Gapped Code Clones Using Gap Locations, in: Proceedings 9th Asia-Pacific Software Engineering Conference, APSEC 2002, pp. 327–336 (2002).

[113] V. Wahler, D. Seipel, J. Gudenberg and G. Fischer, Clone Detection in Source Code by Frequent Itemset Techniques, in: Proceedings of the 4th IEEE International Workshop Source Code Analysis and Manipulation, SCAM 2004, pp. 128-135 (2004).

[114] R. Wettel and R. Marinescu, Archeology of Code Duplication: Recovering Duplication Chains From Small Duplication Fragments, in: Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2005, 8pp. (2005).

[115] T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, S. Kawaguchi and H. Iida, SHINOBI: A real-time code clone detection tool for software maintenance, Technical Report:NAIST-IS-TR2007011, Graduate School of Information Science, Nara Institute of Science and Technology, 2008.

[116] W. Yang, Identifying Syntactic Differences Between Two Programs, SoftwarePractice and Experience, 21(7):739-755 (1991).