Research

On the effectiveness of clone detection by string matching*

Stéphane Ducasse, Oscar Nierstrasz and Matthias Rieger

Software Composition Group, Institute for Applied Mathematics and Computer Science, University of Berne, Neubrückstrasse 10, CH-3012 Berne, Switzerland
This is a preprint of Volume 18, Issue 1 (January/February 2006) On the effectiveness of clone detection by string matching (p 37-58) Stphane Ducasse, Oscar Nierstrasz, Matthias Rieger Published Online: 1 Nov 2005 DOI: 10.1002/smr.317
http://www3.interscience.wiley.com/cgi-bin/jhome/77004487?CRETRY=1&SRETRY=0

SUMMARY

Although duplicated code is known to pose severe problems for software maintenance, it is difficult to identify in large systems. Many different techniques have been developed to detect software clones, some of which are very sophisticated, but are also expensive to implement and adapt. Lightweight techniques based on simple string matching are easy to implement, but how effective are they? We present a simple string-based approach which we have successfully applied to a number of different languages such COBOL**,** JAVA**,** C++**,** PASCAL**,** PYTHON**,** SMALLTALK**,** C **and** PDP-11 ASSEMBLER**. In each case the maximum time to adapt the approach to a new language was less than 45 minutes. In this paper we investigate a number of simple variants of string-based clone detection that normalize differences due to common editing operations, and assess the quality of clone detection for very different case studies. Our results confirm that this inexpensive clone detection technique generally achieves high recall and acceptable precision. Over-zealous normalization of the code before comparison, however, can result in an unacceptable numbers of false positives.**

KEY WORDS: Software maintenance, duplicated code, string-matching, clone detection.

1. Introduction

Duplicated code arises naturally during the development and evolution of large software systems for a variety of reasons. Duplication can have a severely negative impact on the maintenance of such systems due to code bloat, added complexity, missing abstraction, and the need to maintain multiple copies of nearly identical code [1]. Although duplicated code is conceptually simple, it can be surprisingly hard to detect in large systems without the help of automated tools.

Contract/grant sponsor: See acknowledgements section.
†E-mail: {ducasse,oscar,rieger}@iam.unibe.ch

Various approaches have been applied in practice with good results [2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. The main technical difficulty is that duplication is often masked by slight differences: reformatting, code modifications, changed variable names, and inserted or deleted lines of code all make it harder to recognize software clones. One approach to combat this effect is to parse the code and compare the parsed structures.

Although this technique avoids certain problems, it is heavyweight—meaning that it requires a large upfront investment in complicated technology—and, more importantly, it is fragile with respect to differences in syntax, since a parser must be adapted to *every* programming language and dialect under consideration [7]. This is clearly reflected in the following quotation:

> *"Parsing the program suite of interest requires a parser for the language* dialect *of interest. While this is nominally an easy task, in practice one must acquire a tested grammar for the dialect of the language at hand. Often for legacy codes, the dialect is unique and the developing organization will need to build their own parser. Worse, legacy systems often have a number of languages and a parser is needed for each. Standard tools such as Lex and Yacc are rather a disappointment for this purpose, as they deal poorly with lexical hiccups and language ambiguities."* [7].

In summary, many of the approaches [4, 6, 7, 9, 10] are based on parsing techniques and thus rely on having the *right* parser for the right dialect for *every* language that is used within an organization.

Instead, we propose a lightweight approach based on the minimal use of parsing methods and simple string matching for comparison [8]. We cope with differences in formatting by means of a pre-filtering stage that can easily be adapted to different languages and conventions. Sensitivity to smaller changes in the duplicated code segments is targeted by normalizing certain elements of the source code, like identifiers and string constants, using lexical analysis. Comparison by string matching is straightforward, and is readily implemented by standard libraries and tools. Larger changes to duplicated code segments are dealt with by taking gaps into consideration when filtering the result of the comparison.

It has previously been established that string matching offers an effective way to detect duplicated code [8, 11, 12]. The key question, however, is *how good is it* in comparison with more sophisticated approaches? In particular, can a "cheap" solution based on string matching be used to detect the difficult-to-recognize clones that are addressed by the parser-based methods? How do recall (ratio of detected to actual clones) and precision (ratio of detected to candidate clones) of our lightweight approach compare to that of more heavyweight techniques?

We aim to answer this question by applying several variants of the string-matching approach to case studies with varying characteristics. Note that our goal is not to identify clones that can be directly refactored by a refactoring engine but rather to identify code duplication in the larger context of reverse engineering and code understanding. Therefore we favor high recall at the possible cost of low precision.

The contributions of this paper are: (1) a presentation of our approach and and its comparison with related work, (2) the analysis of the impact of normalizations on string-based approaches.

We first describe our line-based string-matching approach (section 3). We review the results of an independent evaluation [12] that indicate that our approach is comparable to that of Baker [2], *i.e.*, it exhibits good recall and average precision (section 4). Next, we consider a variant of our approach

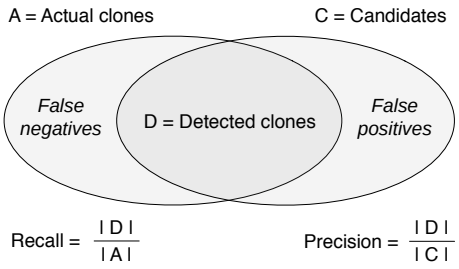

Figure 1. Recall and Precision.

in which code is *normalized* by systematically renaming variables, literals and function names. We consider six different degrees of code normalization and measure their impact on the performance of the string-matching approach to clone detection. We have selected two significant case studies from Bellon's comparative analysis [12] for their different characteristics. The case studies show that, as expected, recall improves at the cost of precision when code is normalized. We also observe that normalizing function names generally reduces precision without any significant improvement in recall. The COOK case study demonstrates that the line-based approach suffers when coding conventions require syntactic elements such as function parameters and arguments to be listed on separate lines. Nevertheless, the case studies offer convincing evidence that a simple string matching approach to detecting duplicated code offers excellent results and performance in return for a minimal investment in tool development.

2. Duplication Detection Challenges

Before we present the details of our string-matching approach to detecting duplicated code, it is important to understand the challenges that any approach must cope with.

Although the notion of duplicated code is intuitively clear, the problem of detecting it is not so well-defined. In Figure 1 we illustrate the key issues. Let us suppose that there exists some ideal set A of *actual clones*, which can only be assessed by inspection. In practice, it is too expensive to determine A by manual inspection, and in any case the results will depend heavily on the *subjective* opinion of the person performing the inspection.

An automated tool will identify some set C of *candidate clones*. Clearly we would like C to be as close as possible to the ideal set A. D is the set of candidates that would reasonably be accepted as being actual clones. *Recall* measures the fraction of actual clones that are identified as candidates, and *precision* measures the fraction of candidates that are actually clones. A good technique should exhibit both high recall and precision, but depending on the context, and the numbers of candidates returned, one might accept, for example, better recall in return for poorer precision.

Consider the following requirements for detecting duplicated code:

Avoid false positives. False positives occur when code is marked as duplicated that should not be. This impacts precision. Programming language constructs, idioms, and recurring statements should not normally be considered as duplicated code, since they do not indicate copy-and-paste problems.

Avoid false negatives. False negatives arise when code is duplicated, but slightly altered in such a way that it is no longer recognized as being a clone. This will impact recall. A good clone detection tool will be robust in the face of insignificant or minor alterations in the duplicated code. The real difficulty is to be precise about when two similar pieces of code should be considered duplicates of one another or not.

Scalability. Duplicated code is most problematic in large, complex software systems. For this reason, a useful tool must be able to cope with very large code bases.

Multiple languages and dialects. There are hundreds of programming languages in use today, and dozens of dialects of the most popular languages (like C++) without counting the language extensions based on macros. A useful duplicated code detector must be robust in the face of syntactic variations in programming languages, and should be configurable with a minimum of effort. In particular, a tool that can only be configured by an expert in parser technology is unlikely to be used.

Some of these challenges conflict with one another. To avoid false negatives we must be able to ignore some of the superficial differences between copied fragments. However, to distinguish the superficial from the essential differences we need some sort of code analysis, *i.e., some* amount of parsing. Whereas deeper code analysis improves the ability to overlook superficial differences to avoid false negatives, it, however, also means increased dependence on a particular language.

3. Our Approach: Detecting Duplicated Code by String matching

Preferring language independence over parsing accuracy, we employ parsers only to the extent that they can be adapted to another programming language by simple reconfiguration. Instead of treating every token in the entire source text we use a *fuzzy* approach to parsing [13] which recognizes only a restricted set of syntactic elements and ignores the rest. Recognition of partial structures is triggered by unique anchor tokens that can be found via regular expression matching, for example. Adaption to another programming language then means merely to configure the anchor tokens.

3.1. A Three Step Approach

Our goal is to arrive at a cheap, effective technique for clone detection that can easily be adapted to new contexts and programming languages, while exhibiting high recall and acceptable precision.

Our approach consists of the following three steps:

1. *Eliminate noise:* Transform source files into *effective files* by removing comments, white space, and other uninteresting artifacts.

We also consider optional code normalizations (Section 3.5) in order to improve recall of clones with slight variations. This is not part of the basic technique, but it is nevertheless a necessary refinement to avoid false negatives.

2. *Compare the transformed code:* Compare effective files line-by-line.
3. *Filter and interpret the results:* Apply some simple filters to extract just the interesting patterns of duplicated code.

False positives are avoided by removing certain kinds of noise and by filtering. False negatives are similarly avoided by removing noise that perturbs the comparisons. The approach scales well (in time) due to its simplicity.

Finally, the approach is easily adapted to different languages since it does not rely on parsing. We have applied it to applications written in COBOL, JAVA, C++, PASCAL, PYTHON, SMALLTALK, C and PDP-11 assembler [14]. The time required to adapt the lexer never exceeded 45 minutes [8]. Noise reduction may have to be adapted for each language, which, since it basically consists in preparing a list of tokens to be removed, is much simpler than adapting a parser.

The results of the clone detection can be reported in various ways. Duploc is a platform that visualizes the comparison matrix as a dotplot [8]. These visual displays are linked to the source code to support navigation in a reverse engineering activity. The results can also be used as input to a refactoring engine that eliminates duplicated code by introducing the missing normalizations [15].

In the rest of this section we provide some details concerning the technique.

3.2. Noise Elimination

Noise elimination serves two purposes. It reduces false positives by eliminating common constructs and idioms that should not be considered duplicated code. It also reduces false negatives by eliminating insignificant differences between software clones.

What should be considered noise depends not only on the programming language, but also on what information you want to extract. Typical operations include:

- eliminating all white space and tabulation to compensate for reformatting
- eliminating comments to improve the focus on functional code, and
- converting all characters to lower case in languages that do not distinguish case.

Other operations that could be performed, depending on the programming language in question, include:

- removing uninteresting preprocessor commands like `#include`
- removing all block and statement delimiters, and
- removing common and uninteresting language constructs like `else`, `break`, or type modifiers like `const`.

Noise elimination is easily specified as a short program in a dedicated text manipulation language, such as Perl [16]. An example for the filtering process is shown in Figure 2.

The noise reduction operations may introduce some false positives since some of the removed entities carry semantics, *e.g.,* block delimiters. Since we generally value false positives over false

```
#include <stdio.h>
static int stat=0;

int main(argc, argv)
    int argc;
    char **argv;
{
  /* skip program name */
  ++argv, --argc;
    if ( argc > 0 ) {
```

```
staticintstat=0
intmain(argc,argv)
intargc
char**argv
++argv,--argc
if(argc>0)
```

Figure 2. Filtering the code snippet on the left yields the result shown on the right.

negatives we do not consider this a serious drawback. We therefore sacrifice some precision to obtain better recall.

The required noise reductions and transformations are implemented with moderate effort. It suffices to build a small tokenizer/lexer with a lookahead of one using regular expressions. In this way, we determine for each token to which class it belongs and to which normalized replacement we have to change it. This approach is generic and is easily adapted for different languages (we have done so for C, C++, JAVA, COBOL, SMALLTALK, PDP 11, and PYTHON). From this perspective, parameterized string matching realized by the combination of normalization and simple string matching is still largely language-independent.

As in any information retrieval task, when increasing recall one has to pay the price of reduced precision (see Section 5). Parameterized string matching not only detects more duplication than simple string matching, but it also produces more false positives.

3.3. Effective Line Comparison

Once we eliminate noise and normalize the code, effective files are compared to each other line-by-line. The naive comparison algorithm is $O(n^2)$, but this is easily improved by hashing lines into B buckets, and then comparing only lines in the same bucket [7].

Lines are compared for exact string matches. The result of the comparison is a matrix of hits and misses that can be visualized as a dotplot [17, 18, 8].

Exact clones are not the norm. Instead, code is more typically duplicated and then modified in various ways. Figure 3 graphically illustrates some typical duplication scenarios: (a) pure duplication results in unbroken diagonals, (b) modified lines appear as holes in the diagonals, (c) deletions and inserts result in broken, displaced diagonals, and (d) case statements and other repetitive structures yield grid-like, rectangular blocks.

The comparison matrix typically contains many hits, some of which are significant, and others which are not. To extract the significant hits, we perform a third, filtering pass.

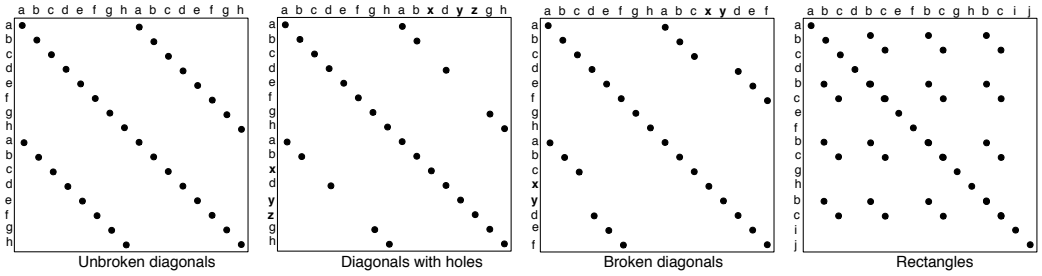

Figure 3. Dotplot visualization of typical duplication scenarios.

3.4. Filtering

In general, we are not interested in individual duplicated lines of code, but in blocks of code that contain "significant" duplication. We call these blocks *comparison sequences*. In order for such blocks to be considered clones, they should be of a "significant" size, and they must contain a "significant" amount of duplication. We therefore quantify the duplication in two comparison sequences by considering their *gap size*, *i.e.*, the length of non-duplicated subsequences. For example, if we compare the sequences "abcduvwefg" and "abcdxyzefg", we find a gap of length 3.

This leads us to the following filter criteria:

1. *Minimum length*. This is the smallest length for a comparison sequence to be considered "significant".
2. *Maximum gap size*. This is the largest allowable gap size for sequences to be considered duplicates of one another.

Filtering is purely an issue of eliminating false positives, since filters only remove candidates identified in earlier phases. The algorithm that performs the filtering is linear in the amount of single matches computed in the comparison phase. Note that we do not detect broken diagonals. Ueda [19] has proposed an $O(n \log n)$ algorithm for this problem.

3.5. Code Normalization to Improve Recall

Although simple string matching can detect a great deal of duplication, it is clear that, in certain cases, it will miss code that has been edited after being copied. Parsing code and comparing the resulting abstract syntax trees, however, is an approach that is more heavyweight than we wish to consider, for reasons we have already outlined.

Instead, we propose to add a second pre-processing phase which *normalizes variable language constructs*, and then performs simple string matching on the transformed programs. A simple way to achieve this normalization is to use regular expressions to match elements of the source code that belong to a common syntactic category, and replace them by a string representing that category. For

Table I. Normalization Operations on Source Code Elements

Operation	*Language Element*	*Example*	*Replacement*
1	Literal String	`"Abort"`	`"..."`
2	Literal character	`'y'`	`'.'`
3	Literal Integer	`42`	`1`
4	Literal Decimal	`0.314159;`	`1.0`
5	Identifier	`counter`	`p`
6	Basic Numerical Type	`int, short, long, double`	`num`
7	Function Name	`main()`	`foo()`

```
def manage_first(self, selected=[]):
  options=self.data()
  if not selected:
    message="No views to be made first."
  elif len(selected)==len(options):
    message="No making all views first."
  else:
    options=self.data()
```

```
def manage_first(P, P=[]):
  P=P.data()
  if not P:
    P="..."
  elif len(P)==len(P):
    P="..."
  else:
    P=P.data()
```

Figure 4. Python source code from the Zope Application Server before and after normalizing all identifiers except function names.

example, all identifiers may be replaced by a generic name like `p` [2, 11]. See Table I for a list of other code elements that can be normalized. In section 5 we analyze the impact of certain combinations of these normalization operations.

Note that in most languages it is simple to distinguish between function names and variable names by the `(` token starting the parameter list of a function declaration or invocation. The reason for not normalizing the keywords of the language is the assumption that language keywords, which give the code its basic structure, must be the same in two code fragments if they are to be considered clones.

We can now define different degrees of normalization by selecting which elements to normalize. A particularly useful degree is that which normalizes every element in Table I except function and method names. See Figure 4 for an example of this kind of transformation.

Now that our approach is described, we analyze how it compares with other approaches.

4. Bellon's Comparative Study

The large variety of clone detection techniques that have been developed in recent years has spurred interest in comparing their effectiveness. Bellon [12] has conducted a comparative case study with

the goal of establishing the relative advantages and disadvantages of the different approaches. We summarize here Bellon's results concerning the string-matching approaches.

4.1. Bellon's Experimental Setup

Reference Set Construction. To compare the different approaches, Bellon built a reference set by manually confirming a set of candidates to be clones. However, it is important to note that this reference set was (1) based on candidates identified by tools participating in the comparison, and (2) incomplete — due to time constraints, Bellon was only able to cover 2% of the candidates.

Clone Types. For his study Bellon categorized all clones into three types: Exact clones (Type 1), clones where identifiers have been changed (Type 2) and clones where whole expressions or statements have been inserted, deleted or changed (Type 3).

Mapping Clone Candidates to References. To decide which candidates correspond to a confirmed clone, Bellon defined a matching criterion based on the notion of *distance* between clones. This criterion assesses a clone pair to be a "good enough" match of another clone pair if the overlap between the two corresponding source fragments is large enough and they are of comparable size. The OK and GOOD metrics [12] determine how well two clone pairs overlap each other, *i.e.,* if they can be declared as *"similar"*. Bellon has used a threshold of 0.7 to determine which candidate clones map to a reference.

4.2. String Matching as Evaluated by Bellon

To evaluate our approach, we provided Bellon with results obtained from non-normalized source code, allowing for a gap of 1 line between matching lines. In the summary of his study [12], Bellon forms two coarse categories of tools: the ones achieving high recall with low precision on the one hand, and the ones sacrificing recall for an improved precision on the other hand. He groups our approach together with the other string- and token-based approaches of Baker [2] and Kamiya [11] in the former category, and notes that we are closest to Baker's in terms of number of reported and rejected candidates.

More detailed results from Bellon's study are shown in the next table. We list the differences of Baker and Kamyia to our own, setting our values as 100%.

Differences to Rieger	WELTAB		COOK	
	Baker	Kamyia	Baker	Kamyia
Retrieved Candidates	+988	+2144	-113	-6318
References matched with OK	+26	-66	-2	-105
References matched with GOOD	+50	-97	-1	-42
Precision	-0.003	-0.008	0	+0.03

Regarding the number of returned clone candidates, our approach was closest to Baker's which is normal since both approaches make line breaks a factor of comparison.

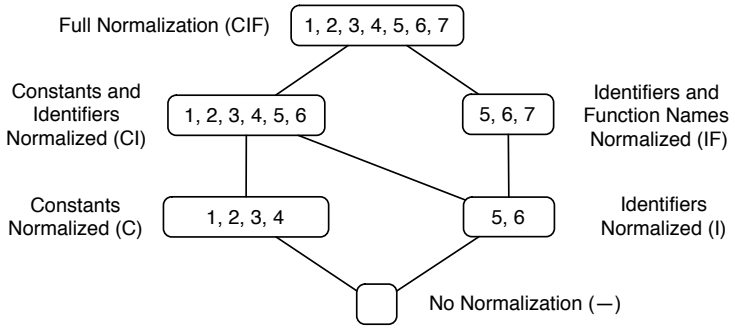

Figure 5. Different degrees of normalizations and their relationships.

5. Assessing the Impact of Normalization

When participating in Bellon's case study, we compared only non-normalized code. We now wish to consider the impact of normalizing source code (Section 3.5) on the effectiveness of our technique. It is clear that normalizing code elements increases the number of single matches, as the diversity among the source lines is reduced. The question is how these additional single matches affect recall and precision for the clones retrieved by the filtering (Section 3.4). In particular, we want to assess to what extent can recall be improved without sacrificing precision. We answer this question by exploring the impact of various degrees of normalization on the quality of clone detection by string matching for two representative systems from the Bellon set.

For each system we performed the following steps: (1) select the normalization operations and transform the code, (2) compute the candidate clones, and (3) compare the candidate set to a reference set to assess precision and recall.

5.1. Degrees of Normalization and Gap Size

To perform this evaluation we define and assess six degrees of normalization as shown in Figure 5. Independently we assess the impact of varying the gap size.

5.1.1. Degrees of Normalization

By *code normalization* we mean replacing certain elements of a program with generic placeholders with the aim of only removing inessential information (see Table I). The transformed code should still represent essential features which are then better recognized in the comparison. We define six degrees of normalization that make use of various subsets of the normalization operations listed in Table I. These six degrees form a lattice, illustrated in Figure 5, reflecting which normalization operations are performed. These normalization degrees correspond to the different editing operations a programmer may perform when duplicating code.

No Normalization (–). Here, only the basic noise reduction as described in Section 3.2 is applied to the source code. The results gathered for this degree demonstrate the effectiveness of the basic approach.

Constants normalized (C). In addition to the noise removal we normalize literal strings, characters and numerical constants, *i.e.*, we mapped them all to a similar token (operations 1, 2, 3 and 4 of Table I).

Identifiers normalized (I). After noise removal we normalize some lexical language elements: identifiers, labels, basic numeric types (operations 5 and 6).

Identifiers and Function names normalized (IF). In addition to identifiers, we change all function names in declarations and invocations to `foo` (operations 5, 6, and 7).

Constants and identifiers normalized (CI). This includes all operations except function name normalization.

Full Normalization (CIF). Here we apply all the normalization operations.

5.1.2. Gap Size

The filtering of clone candidates as described in Section 3.4 can be thought of as an *ad hoc* normalization mechanism. Indeed, a gap in a sequence of matching lines occurs when corresponding lines fail to match. When we allow a gap in a sequence of matching lines, we are normalizing the contents of these lines. Because of the generality of this mechanism, the increase of noise or the loss of precision due to it is noticeable.

In this experiment we want to compare the gap mechanism with the degree of normalizations. The gaps in a comparison sequence can be controlled by the *maximum gap size* filter criterion. We have set this parameter to the values 0,1, and 2. Based on our experience, we choose to let no more than 2 consecutive non-matching lines in a comparison sequence of minimal length 6 which is the same minimal length as was agreed upon by the participants of the Bellon study. Our experiment generates then 18 different data sets based on the six different normalization degrees and the 3 gap sizes.

5.2. Case Study Selection

We selected the case studies to cover the following criteria: (a) availability of reference data for other clone detection tools, (b) differences in coding style and line layout, (c) real applications developed by external persons, and (d) common programming language, to avoid possible influence of programming paradigms. (Note that we have separately carried out experiments with case studies from different programming languages, to assess the language-independence of the approach [14].)

We chose two applications from Bellon's comparative study: the WELTAB application consisting of 39 files (9847 LOC) and the COOK application consisting of 295 files (46645 LOC) [12]. WELTAB is a relatively small application known to contain considerable duplicated code, and is therefore convenient for carrying out experiments. The COOK application adopts a code formatting approach in which

function arguments and parameters are put on separate lines, posing a special challenge to line-based clone detection approaches.

Because of time constraints we were not able to investigate all eight systems from Bellon's study at the required level of detail. Note however that for the two case studies, we *manually* evaluated a large portion of the clone candidates. For the other six systems we provide in an Appendix numbers that can be computed without manual intervention.

5.3. Construction of the Reference Set

To compute recall and precision, we need to compare the candidate clones reported by our tool with a reference set of validated clones. We have constructed such a reference set from two sources: (1) the (incomplete) reference set by Bellon [20] which was assembled by manually examining candidates detected by various tools, and (2) the result of a manual evaluation of the results reported by our tool as shown by the following table.

Case Study	Retrieved Candidates	Evaluated Candidates	Confirmed Clones
WELTAB	10392	8411	6499
COOK	82655	46288	5672

To be clear: we measure recall using the confirmed clones that Bellon selected from the candidates of all tools participating in his study. Precision, on the other hand, will be established using the confirmed clones reported by our own tool.

Note that, although the reference set is not homogeneous, and not necessarily complete, one can still obtain meaningful figures for recall and precision. To assess recall, one may use an arbitrary, sufficiently large set of confirmed clones. The reference set need not be complete. To assess precision, it suffices to manually examine a representative sample of the candidate clones detected. To determine the recall rate of our tool we have mapped the retrieved candidates to the reference clones by way of the mapping function defined by Bellon [12] with the same *OK* threshold of 0.7.

6. Results

We now summarize the results of our experiments. We present the numbers of candidate clones detected, recall for different categories of clone types, precision, and performance.

6.1. Retrieved Candidates

Table II shows the number of the candidates identified with the largest setting of the maximum gap size. In Figure 6 we plot the percentage increase in the number of identified candidates that normalization brings, in comparison to clone detection without normalization. As expected, COOK shows a more-or-less steady increase in candidates identified as more normalization operations are applied. In WELTAB, however, we notice a considerable but puzzling jump when constants and identifiers are normalized jointly.

Table II. Retrieved Candidates for the different normalization degrees (maximum gap size = 2)

	Normalization degree	WELTAB *Candidates*	COOK *Candidates*
–	No Normalization	1467	7661
C	Constants normalized	2255	12434
I	Identifiers normalized	2565	29333
IF	Identifiers, function names normalized	2608	38141
CI	Constants, identifiers normalized	5946	38581
CIF	Full Normalization	6334	49789

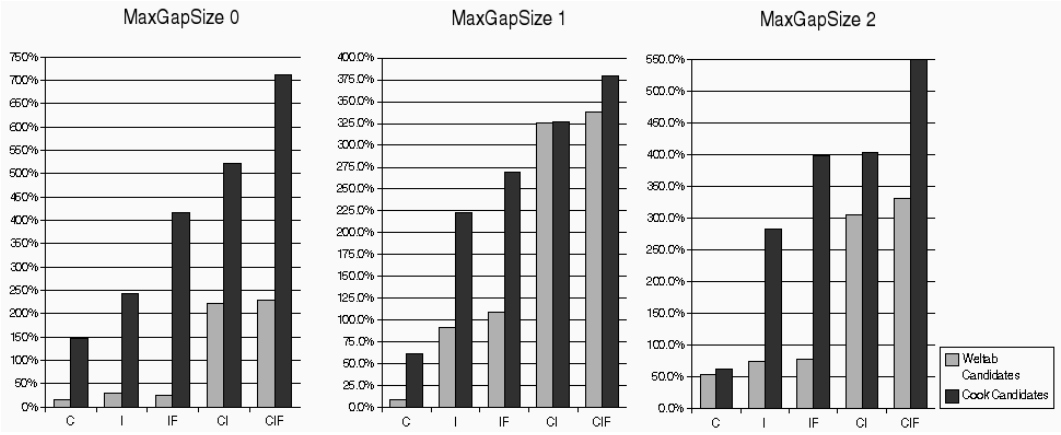

Figure 6. The increase in retrieved candidates, measured relative to the lowest degree of normalization.

6.2. Recall by Clone Types

The effectiveness of a clone detection techniques will vary depending on how much a clone has been edited after copying. We adopt Bellon's classification of clone types in an effort to measure recall as a function of both editing operations and degrees of normalization.

In the WELTAB case study (see Figure 7), we see that overall recall (all types) increases from 78.2% to 95% when introducing more normalization. For Type 1 clones (identical clones), the recall rate is 100% at all degrees, as would be expected from exact string matching.

The lowest recall for non-normalized code is registered for type 2 clones (renamed identifiers). This can be explained by the observation that identifier changes are likely to occur systematically on most of the lines of a clone. Exact string matching will therefore miss every line thus modified, and consequently fail to identify the clone. With the normalization transformation, however, the recall rate rises by a remarkable 25% to a final level almost equal to that for type 1 clones. It does not reach 100%

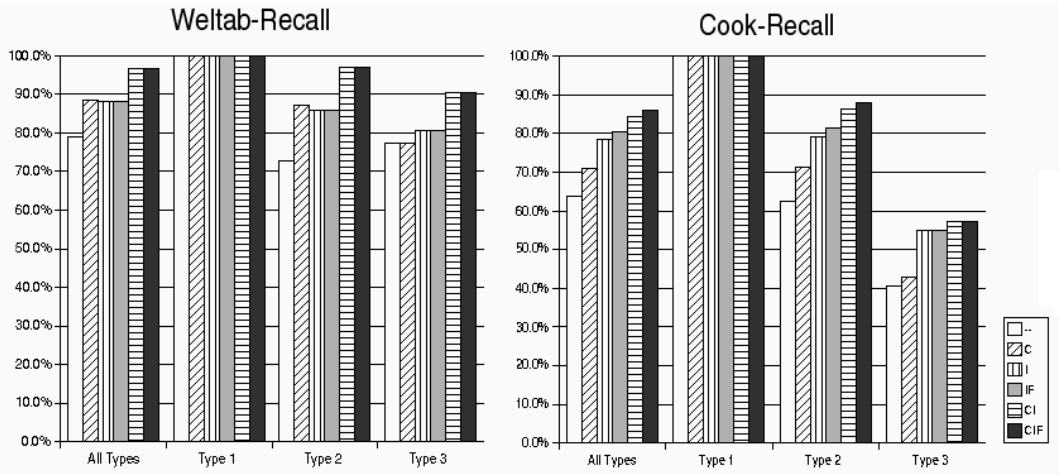

Figure 7. Recall for WELTAB and COOK split by clone type. Maximum gap size is 2.

because we normalized constants and identifiers with different tokens which fails in the case when a constant parameter has been changed into a variable.

For type 3 clones (arbitrary edits), recall is initially higher than for type 2 clones. This can be explained by the fact that we take gaps (non-matching lines) into account when collecting the clones. However, since the normalization operations we perform are local to a single line, we cannot improve detection rates as much as for type 2 clones. Nevertheless, we achieve a recall rate of 90% at the highest normalization degree.

In the COOK case study, the picture is similar. For type 1 clones, recall is 100% already at the lowest degree of normalization. For type 2 clones, recall rises from 63% to 88%, again by about 25%. Type 3 recognition is the worst for all degrees of normalization.

We investigated why the recall was so low for COOK type 3 clones and found a number of reasons:

- code that was syntactically too different to be recognized as a clone,
- altered source elements that we did not normalize, *e.g.*, type casts, pointer dereferences,
- altered formatting of source lines,
- source text inserted or deleted from the middle of a clone, and
- clones too small to be retrieved by our specification.

In Figure 8 we see how recall varies in response to increasing degrees of normalization. Increasing the maximum gap size from 0 to 1 improves recall significantly, whereas a maximum gap size of 2 has less impact. Normalizing constants improves recall for both WELTAB and COOK, whereas normalizing identifiers and function names is good only for COOK.

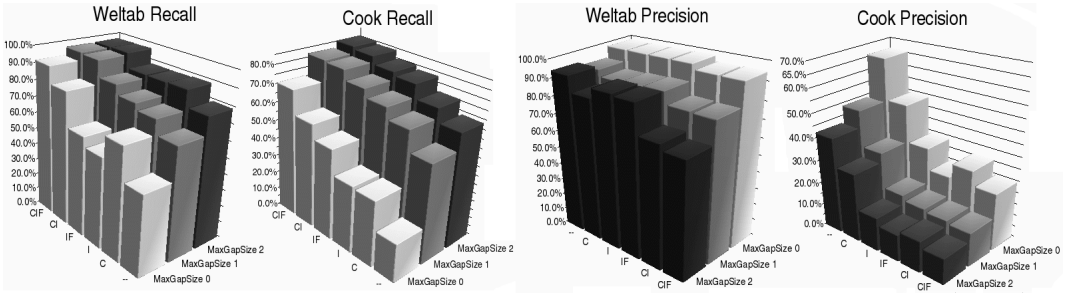

Figure 8. Recall and precision for WELTAB and COOK.

6.3. Precision

We now consider how precision varies with respect to the degrees of normalization. The studies illustrate well the phenomenon that precision diminishes with increasing recall.

With the WELTAB case study, we observe a precision of 94% for non-normalized code, but this drops to 70% at the highest degree of normalization. The very high precision of WELTAB is consistent with the results of Bellon's experience that the confirmation rate for WELTAB candidates (coming from all the participating tools) was, at 90%, the highest among all the systems under study.

The situation is not so good in the COOK case study, where initial precision (*i.e.*, without normalization) is only 42% and drops to 11.5% for the highest degree of normalization. The latter clearly represents an unattractive level of noise for an engineer who is searching for refactoring opportunities.

A similar drop in precision from the WELTAB to the COOK case studies was reported by Bellon for both our approach and that of Baker.

6.4. Time and Space Analysis

The prototypical detector used for this case study, Duploc, is implemented in VisualWorks 7.1 Smalltalk as an exploratory and flexible platform. Providing optimal speed was not the guiding factor in its design. We measure the time and space required for different phases of our approach using a 2.1 GHz AMD Athlon with 550 Mb of memory running Linux 2.4.

Evaluating precisely the amount of space required is difficult since it is based on the chosen representation. In our tool, we keep all the information in memory as objects in a Smalltalk image. In the following table the space number indicates only the space required for storing the computed duplication, not for the representation of the source code since this does not change much when normalizing the code. High normalization degrees make the code more uniform which increases the number of single matches. This is the reason for larger memory requirements seen in COOK. Note that the comparison and filtering phases are mostly determined by the number of single matches.

Case Study	Normalization Degree	# of Matches	Comparison	Filtering	Space
WELTAB	–	260,710	5.8s	0.7s	2.2MB
	C	388,732	6.4s	0.9s	2.4MB
	I	365,289	7.0s	0.8s	3.2MB
	IF	429,226	7.3s	0.8s	3.4MB
	CI	649,817	8.8s	1.1s	3.8MB
	CIF	726,470	9.5s	1.2s	4.2MB
COOK	–	1,485,630	150s	15.8s	59MB
	C	2,166,832	183s	22.8s	88MB
	I	5,771,343	469s	61.3s	252MB
	IF	6,825,959	690s	64.3s	330MB
	CI	6,629,632	546s	63.7s	289MB
	CIF	7,703,927	750s	88.8s	380MB

6.5. Conclusions

The experiment clearly demonstrates the limited value of normalizing function names. In the WELTAB case study, recall did not improve at all, but precision dropped by about 3%. With COOK, recall improved by about 1.3%, while precision dropped by 2.5%. Since for COOK the reduced precision meant 11,000 more candidates were retrieved at the highest degree of normalization (gap size 2), the impact of normalization can be quite significant.

In both case studies, normalizing identifiers yields a gain in recall with a corresponding loss in precision. Since recall without normalization is already quite high, this suggests that normalization operations are of very limited value.

We also note that setting a maximum gap size of one rather than two improves precision by 10%, while impacting recall by no more than 5%. We conclude that setting a general gap size of two is not worthwhile, noting however that with increasing lengths gap sizes of 2 will probably become less detrimental to the relevance of a clone candidate.

The experiment also shows that our clone detection technique gives very different results for different case studies. The unusual layout of the COOK case study is especially problematic for simple line-based string-matching. Whereas WELTAB adopts a very traditional code layout, COOK has a sparse code layout where function calls are often spread over multiple lines. Lines then contain only one identifier which leads, especially if identifiers are normalized, to a great number of spurious matches and consequently to a drop in precision.

7. Comparison with Other String-Based Approaches

Many different techniques have been applied to identify copy and paste plagiarism [21, 22, 23, 24]. Techniques used include: structural comparison using pattern matching [4], metrics [5, 6], statistical analysis of the code, code fingerprints [25, 26, 3], or AST matching [7], slicing of program dependence graphs (PDGs) [10],[27], or a combination of techniques [28]. The approaches of Johnson [3], Baker [2], Kamiya *et al.* [11], and Cordy *et al.* [29] are based on string matching and are closest to ours.

Johnson [3], Baker [2], and Kamiya *et al.* [11] employ simple lexical transformations, and Cordy *et al.* [29] use pretty-printing to prepare the source code for clone detection. The main differences amongst these approaches are the selection of the *comparison granularity*, the choices regarding *code normalization*, and the *comparison algorithm*:

- Johnson [3] does not change the source text at all. He creates the comparison element by building *ngrams*. However, Johnson did not participate in Bellon's comparative study.
- Baker [2] selects a single line as the grain of comparison, just as we do. She argues that program editors and programmers work in a line-oriented way. She introduces a mechanism to normalize identifiers which respects the local context in which identifier names are changed. Two code sections are jointly recognized as duplicates if their identifiers have been systematically replaced, *i.e.,* `x` for `width` and `y` for `height`. This prevents some of the false positives that our more simple approach produces (see below for an investigation of this technique). She uses a suffix tree algorithm for comparison.
- Kamiya *et al.* [11] work at the granularity level of individual tokens. They perform many code normalizations in the same manner as we do, however they do not analyze the impact of individual normalization operations. They employ a suffix tree comparison algorithm similar to Baker's.
- Cordy *et al.* [29] use an island grammar to identify syntactic constructs in code. These are then extracted and used as smallest comparison unit. The code is pretty-printed to isolate potential differences between clones to as few lines as possible, but there is no normalization performed at all. The comparison is done using the UNIX diff tool which makes it line based.

7.1. Comparative Studies

Now we compare our approach with that of Baker and Kamiya. Since both have participated in Bellon's study, comparative data are available. We have already summarized Bellon's analysis of the string matching approaches in Section 4.2. Here we wish to investigate how the approaches of Baker and Kamiya compare against different degrees of normalization. We chose the best result of Kamiya *i.e.,* the voluntary submission where some noise was removed by Kamiya. For our tool we did not remove any pieces of code that would favor us.

To obtain a meaningful comparison, we first determine which choice of minimum gap size and normalization degree returns a comparable number of candidates. Then we analyze recall and precision for this specific configuration.

Case Study	Data	Baker	Rieger		Kamiya	Rieger
WELTAB	Candidates	**2742**	2378 (IF, 1)	2608 (IF, 2)	**3898**	3761 (CIF, 0)
	Recall	80%	86%	88%	93%	92%
	Precision	80%	90%	90%	99%	91%
COOK	Candidates	**8593**	9043 (CIF, 0)	7661 (–, 2)	**2388**	2764 (C, 0)
	Recall	70%	71%	64%	43%	36%
	Precision	29%	26%	42%	42%	49%

Contrary to expectations, Baker exhibits somewhat worse precision for WELTAB. For COOK, her precision is slightly better than ours, though we can significantly improve precision at a 7% cost in recall. We partly attribute Baker's loss in precision to some noise (`#include`) that she does not remove.

Kamiya on the other hand exhibits a better precision than we do for comparable recall in WELTAB. In COOK, his recall is better but our precision is better for a comparable number of candidates.

In a second analysis, we identify configurations of our tool that exhibit similar precision to the other approaches, and then we compare recall and the number of retrieved candidates.

Case Study	Data	Baker	Rieger	Kamiya	Rieger	
WELTAB	Precision	**80**%	82% (CIF, 1)	**99**%	98% (IF, 0)	
	Recall	80%	96%	93%	61%	
	Candidates	2742	4973	3898	1414	
COOK	Precision	**29**%	30% (C, 2)	**45**%	42% (–, 2)	49% (C, 0)
	Recall	70%	89%	43%	64%	36%
	Candidates	8593	2255	2388	7601	2764

From these numbers a consistent ranking cannot be derived. We see that our simple approach can achieve results similar to the other two in all cases. For WELTAB, normalizing identifiers and function names seems to be important to obtain similar results. For COOK, however, normalizing identifiers results in too many candidates. We must therefore restrict ourselves to normalizing constants only, or setting the maximum gap size to 0. The set of applicable normalization operations is thus shown to depend on the system under study.

Both Baker's and Kamyia's approaches being string matching techniques like ours, they differ only in a few points from our proposed approach. We will now take a closer look on a key difference for each approach to find out how they influence the results.

Systematic Identifier Normalization: Baker does not replace names of identifiers indiscriminately by one and the same token, but makes the consistent replacement of identifiers one criterion of the comparison, thereby avoiding clone candidates which have the same syntactic structure but differing identifier usage.

By filtering out candidate clones where identifiers are mapped inconsistently (according to the description of Baker in [2]) we can derive how much precision is lost when uniformly normalizing identifier names. At gap size 0 we get the following percentages of candidates which exhibit inconsistent identifier mapping:

Normalization Degree	WELTAB	COOK
–	0.0%	0.0%
C	0.0%	0.1%
I	1.0%	5.8%
IF	1.1%	7.3%
CI	3.8%	7.3%
CIF	3.8%	9.6%

The more we normalize identifiers the more inconsistency is naturally found. Filtering ten percent of the retrieved candidates could certainly be interesting if they all would prove to be false positives. The merit of using this characteristic as filtering criterion is however less clear for clones where the two copied fragments are more distant, as can be found among the results of, for example, metrics based methods. Finding inconsistently mapped identifiers is then no longer very effective at spotting false positives. From the confirmed clones of Bellon's study we can flag as having inconsistent identifier mappings 20.2% of WELTAB and 20.5% of COOK references.

Token Based Comparison: Kamiya, rather than using source lines, compares the code on the granularity level of tokens. This avoids problems with *line break relocation* where only the layout of the code is changed (this problem can also be adressed by *pretty-printing* the code since such tools are readily available for many languages). The smaller granularity however also means that more entities must be processed.

When investigating the references that were detected only by Kamiya but not by us, there was only one example of a clone where layout changes prevented the line-based comparison from detection. This fact can however not be generalized, since it is influenced by the particular construction of the reference set and the characteristics of the case studies. In their own investigations Kamiya *et al.* [11] have reported that as much as 23% of the clones found by the token-based comparison exhibited line break relocation.

8. Conclusions and Future Work

We have presented a lightweight technique to identify duplicated code based on simple string matching. We have reported the results of an external comparative study which classifies our approach with those of Baker [2] and Kamiya [11] as exhibiting high recall and average precision. We have carried out a more extensive study into the impact of code normalization on recall and precision.

Our evaluation shows that allowing for some variation in duplicated code is necessary to get decent recall. We were not able to conclude that gaps in clones or specific normalization of certain source elements is better. A maximum gap size of 1 yields good results, but allowing two lines as gaps can lead to an undesirable loss in precision. However, a similar drop in precision can result from aggressive normalization. In particular, normalizing function names can lead to a significant loss of precision that is not worth the minimal gain in recall.

We have also demonstrated that more sophisticated approaches, such as parameterized matches [2], offer only small advantages over the lightweight approach based on simple string matching.

Our approach has the advantage of being largely language independent. We have successfully applied it to applications written in COBOL, JAVA, C++, PASCAL, PYTHON, SMALLTALK, C and PDP-11 assembler [8] [14], while never requiring more than 45 min to adapt the noise elimination filter to a new language.

One of the problems with the simple approach we promote is that large numbers of false positives can be generated for certain systems with unusual formatting. Future work should address ways of dealing with this. With *increased lexical analysis* we can, for example, normalize literal arrays, a source of many false positives in COOK, or filter clones that cross function boundaries. Another promising direction is to cluster all $n(n-1)/2$ clone pairs that are produced by n instances of the same source fragment into *clone classes*. This will reduce the number of instances that have to be investigated individually.

We deliberately chose the COOK case study for its unusual layout conventions. Our study has confirmed that line-based string matching approaches are sensitive to layout conventions. The strong variation between the two case studies suggests that future work in clone detection should focus not only on the detection of duplicated code but also on the analysis of variables that are outside the code itself such as the coding style, the programming language, the development process, or the programmer's education. Understanding these factors would make the value of the various approaches more quantifiable.

Table III. Other Systems of the Bellon Study

CaseStudy	Normalization Degree	Comparison	# of Candidates	Recall
NETBEANS-JAVADOC	–	3s	198	49.1%
	CIF	15s	2804	87.3%
ECLIPSE-ANT	–	5s	201	56.7%
	CIF	30s	2099	90.0%
SWING	–	180s	5561	69.9%
	CIF	700s	34659	85.3%
ECLISPE-JDTCORE	–	240s	18438	54.9%
	CIF	1380s	60314	74.6%
POSTGRESQL	–	300s	20294	66.3%
	CIF	3000s	41083	71.5%
SNNS	–	180s	29767	50.3%
	CIF	5400s	88565	80.8%

APPENDIX: OTHER SYSTEMS OF THE BELLON STUDY

The Bellon case study comprised six more systems (Figure III) that were to be searched for clones. Whereas Bellon has reviewed manually 2% of the clone candidates found by the study's participants in all eight systems, we have used only two of the eight systems, manually assessing 80% (WELTAB) and 56% (COOK) of the clone candidates retrieved by our tool. To give an impression of how our approach performs with the other systems we present numbers that we could compute automatically without manual intervention.

We let our detector run on all six systems, setting the normalization degrees to the extremes (–) and (CIF) and fixing gap size at 2. We give the time the comparison took (in seconds), the number of clone candidates retrieved, and the recall ratio with regard to Bellon's reference set for that system, computed again using the mapping function with the *OK* threshold of 0.7. Since Bellon's reference sets are not complete, precision can not be evaluated without manually investigating the clone candidates, which we refrained from due to time constraints.

ACKNOWLEDGEMENT

We gratefully acknowledge the financial support of the Swiss National Science Foundation (SNF) and the Swiss Federal Office for Education and Science (BBW) for the projects "Framework-based Approach for Mastering Object-Oriented Software Evolution" (FAMOOS), ESPRIT Project 21975 / Swiss BBW Nr. 96.0015, "A Framework Approach to Composing Heterogeneous Applications", Swiss National Science Foundation Project No. 20-53711.98, and "Meta-models and Tools for Evolution Towards Component Systems", Swiss National Science Foundation Project No. 20-61655.00 and Recast: "Evolution of Object-Oriented Applications" 2000-061655.00/1

REFERENCES

1. Demeyer S, Ducasse S, Nierstrasz O. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, San Fransisco CA, 2002; 282 pp.
2. Baker B. On finding duplication and near-duplication in large software systems. In *Proceedings Second Working Conference on Reverse Engineering*, IEEE Computer Society: Los Alamitos CA, 1995; 86–95.
3. Johnson J. Substring matching for clone detection and change tracking. In *Proceedings International Conference on Software Maintenance (ICSM 1994)*, IEEE Computer Society: Los Alamitos CA, 1994; 120–126.
4. Paul S, Prakash A. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering* 1994 **20**(6):463–475.
5. Mayrand J, Leblanc C, Merlo E. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings International Conference on Software Maintenance 1996*, IEEE Computer Society: Los Alamitos CA, 1996; 244–253.
6. Kontogiannis K. Evaluation experiments on the detection of programming patterns using software metrics. In *Proceedings Fourth Working Conference on Reverse Engineering (WCRE '97)*, Baxter I, Quilici A, Verhoef C (eds.). IEEE Computer Society: Los Alamitos CA, 1997; 44–54.
7. Baxter I, Yahin A, Moura L, Sant'Anna M, Bier L. Clone detection using abstract syntax trees. In *Proceedings International Conference on Software Maintenance, 1998*, IEEE Computer Society: Los Alamitos CA, 1998; 368–377.
8. Ducasse S, Rieger M, Demeyer S. A language independent approach for detecting duplicated code. In *Proceedings IEEE International Conference on Software Maintenance (ICSM '99)*, Yang H, White L (eds.). IEEE Computer Society: Los Alamitos CA, September 1999; 109–118.
9. Balazinska M, Merlo E, Dagenais M, Laguë B, Kontogiannis K. Partial redesign of java software systems based on clone analysis. In *Proceedings Sixth Working Conference on Reverse Engineering*, Balmas F, Blaha M, Rugaber S (eds.). IEEE Computer Society: Los Alamitos CA, 1999; 326–336.
10. Krinke J. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering (WCRE'01)*, IEEE Computer Society: Los Alamitos CA, 2001; 301–309.
11. Kamiya T, Kusumoto S, Inoue K. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 2002; **28**(6):654–670.
12. Bellon S. Vergleich von Techniken zur Erkennung duplizierten Quellcodes. *Master's thesis*, Institut für Softwaretechnologie, Universität Stuttgart: Stuttgart Germany, 2002; 156 pp.
13. Koppler R. A systematic approach to fuzzy parsing. *Software—Practice and Experience* 1996; **27**(6):637–649.
14. Rieger M, Ducasse S, Nierstrasz O. Experiments on language independent duplication detection. *Technical Report iam-04-002*, Institute of Applied Mathematics and Computer Science, University of Bern: Bern Switzerland, 2004; 30 pp.
15. Koni-N'sapu G. A scenario based approach for refactoring duplicated code in object oriented systems. *Diploma thesis*, Institute of Applied Mathematics and Computer Science, University of Bern: Bern Switzerland, 2001; 110 pp.
16. Wall L, Christiansen T, Orwant J. *Programming Perl* (3rd edn). O'Reilly & Associates, Inc., Sebastopol CA, 2000; 1092 pp.
17. Gibbs A, McIntyre G. The diagram: A method for comparing sequences. Its use with amino acid and nucleotide sequences. *European Journal of Biochemistry* 1970; **16**:1–11.
18. Helfman J. Dotplot patterns: A literal look at pattern languages. *Theory and Practice of Object Systems (TAPOS)* 1995; **2**(1):31–41.
19. Ueda Y, Kamiya T, Kusumoto S, Inoue K. On detection of gapped code clones using gap locations. In *Proceedings Ninth Asia-Pacific Software Engineering Conference (APSEC'02)*, IEEE Computer Society: Los Alamitos CA, 2002; 327–336.
20. Bellon S. Detection of software clones. Institut für Softwaretechnologie, Universität Stuttgart: Stuttgart Germany, 2002. Http://www.iste.uni-stuttgart.de/ps/clones/ [27 April 2005].
21. Halstead M. *Elements of Software Science*. Elsevier North-Holland, 1977.
22. Grier S. A tool that detects plagiarism in Pascal programs. *ACM SIGCSE Bulletin* 1981; **13**(1):15–20.
23. Madhavji N. Compare: A collusion detector for Pascal. *Techniques et Sciences Informatiques* 1985; **4**(6):489–498.
24. Jankowitz H. Detecting plagiarism in student Pascal programs. *Computer Journal* 1988; **1**(31):1–8.
25. Manber U. Finding similar files in a large file system. In *USENIX Winter 1994 Technical Conference Proceedings*, USENIX Association: Berkeley CA, 1994; 1–10.
26. Johnson J. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: Software Engineering - Volume 1*, Gawman A, Kidd E, Larson P (eds.). IBM Press, 1993; 171–183.
27. Komondoor R, Horwitz S. Eliminating duplication in source code via procedure extraction. *Technical Report 1461*, Computer Sciences Department, University of Wisconsin-Madison: Madison WI, 2002; 10 pp.
28. Balazinska M, Merlo E, Dagenais M, Laguë B, Kontogiannis K. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings Seventh Working Conference on Reverse Engineering (WCRE'00)*, Balmas F, Kontogiannis K (eds.). IEEE Computer Society: Los Alamitos CA, 2000; 98–107.

29. Cordy J, Dean T, Synytskyy N. Practical language-independent detection of near-miss clones. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research (CASCON 04)*, IBM Press, 2004; 1–12.