Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics

K. Kontogiannis
University of Waterloo
Dept. of Electrical & Computer Engineering
Waterloo, ON. N2L 3G1
Canada

Abstract

Cloning of code fragments in large systems is a common practice that may result in redundant code, higher maintenance costs, and less modular systems. This paper examines and evaluates the use of five data and control flow related metrics for identifying similar code fragments. The metrics are used as signatures for a code fragment. Matching on such signatures results in fast matching that can be used to locate instances of code cloning even in the presence of modifications such as changes in variable names, and insertion of statements. The paper takes an information retrieval approach and reports on experiments conducted for retrieving code fragments in three different software systems.

1 Introduction

The localization of programming patterns and the identification of code cloning instances are two important tasks for the re-engineering or the remediation of a large software system. Cases where the identification of similar code fragments is useful include cases of plan recognition for program understanding or Y2K remediation [Wills93], [Rugaber96], system partitioning where code cloning can be used as a clustering criterion, software migration to object-oriented environments where instances of cloned code can be abstracted to class methods [Bowdidge96] and finally, in software maintenance where similar corrections/enhancements may have to be applied to all instances of similar algorithms in the system [Baker95].

A number of research teams have investigated the issue of code cloning. In [Baker95] a string based approach is proposed where exact or parameterized matching is achieved based on a modification of the Bayes-Moore algorithm. In [Johnson96] a string-based approach which is based on signatures calculated from the program text allows for exact matching on large software systems. In [Mayrand96], a clone detection system using metrics is presented. That system allows for detecting clones according to specific categories of comparison (control flow, layout, informal information). In [Wu92], [Muth] a Unix utility called *agrep* allows for approximate matching between a regular expression-like pattern and text in source code or plain text files. The *agrep* uses a set of constraints in terms of allowable insertions/deletions between the pattern and the code for establishing the limits of the partial matching process. In [Myers89] a partial matching technique based on regular expressions and dynamic programming is proposed. It is similar to the *agrep* utility but allows for more general patterns to be formulated. Finally, in [Konto96] a partial matching stochastic technique that is based on syntactic patterns specified in a abstract query language and Markov models is proposed for identifying similar code structures. That approach allows for the formulation of patterns that contain data type information (i.e. date fields for Y2K remediation), and data flow constraints (i.e. definitions and uses of variables).

The approach discussed in this paper is fast and allows for matching even in the presence of arbitrary local changes in the code. Its drawback is that it may result in lower precision for higher recall rates. However, the experiments discussed in this paper reveal that this technique can be used as a fast approximation for partial code cloning detection, compared to other techniques that either do not allow for such matching parameterization or are computationally expensive.

The distance metric used for the experiments pre-

*This work was funded by IBM Canada Ltd. Laboratory - Center for Advanced Studies (Toronto) and the National Research Council of Canada.

sented in this paper is the Euclidean distance between elements on a 5-dimensional space created by the software metrics considered. Since the objective of our experiments was to observe and evaluate the use of such a metric-based cloning detection technique in the best case scenario, we examined the behavior of the matcher (precision, recall) on results obtained at zero distance values.

Within this framework (zero distance constraint), Euclidean distance is simple to calculate and produces the same set of results as compared to the results obtained using other metric distances. Note that in order to define a distance metric that directly reflects similarity (i.e it is monotonic according to a code similarity ordering criterion), one must first define such a code similarity criterion. In the literature formal approaches to specify program behavior are presented [Stoy77]. However, there are no practical definitions of code similarity that relate to software metrics and therefore it is difficult to define such a distance metric. In this paper (Section 3), we provide a subjective, intuitive set of criteria for code cloning. These criteria conform with the criteria presented in related research studies [Baker95], [Johnson96], [Mayrand96].

The paper first gives an overview of the features used in the matching process, provides the experimentation framework and discusses the results drawn from a number of experiments using this metrics based technique.

2 Features For Cloning Detection

The first step towards analyzing a software system is to represent the code in a higher level of abstraction. A number of program representation schemes have been proposed in the relevant literature. These include frames [Ning94], annotated data and control flow graphs [Wills93], Abstract Syntax Trees [Newcomb97], logic formulas on program dependencies [Canfora94] and, relation tuples based on a language domain model [Muller93].

We have chosen as a program representation scheme, the program's annotated abstract syntax tree (AST). We believe that this scheme is most suitable because: *a)* it does not require any overhead to be computed as it is a direct product of the parsing process and, *b)* it can be easily analyzed to compute several data and control flow program properties.

Nodes of the AST are represented as objects in a LISP-based development environment[1].

The tree is created during the parsing phase, and is annotated in a post-processing phase where linking information as well as a number of features for every statement and function node of the system are computed. The matching features were selected on their contribution to the data and control flow of the system. We aimed for the features to exhibit low correlation (based on the Spearman-Pierson correlation test) so as to be sensitive to different control and data flow properties. The features selected for our analyses include:

1. The number of functions called (Fanout);
2. Global and local variables [2] used and updated;
3. Parameters passed by reference used and updated;
4. Parameters passed by value used and updated;
5. Input/Output operations;
6. External files used;
7. Decision predicates;

These features have been computed compositionally in the AST from simple statements and expressions to composite statements, blocks, and functions as discussed in [Konto96]. The benefit of computing in a compositional manner the features above is that it allows for matching at different levels of granularity and type of occurrence. For example a `Block` statement that is part of the *else-clause* of an `If-Then-Else` statement can be compared with a `Block` statement that is part of the body of a `While` statement and may be found similar if their corresponding feature vectors match.

The reason we have chosen these features is that they provide a global view of both data and control flow properties of a code fragment, and exhibit low correlation values. The correlation values of the features used, computed using the Spearman-Pearson correlation test, are presented in Table.1. Adding more features would be of benefit only if these additional features have low correlation value as compared with the existing features. It is evident that we could have used the features instead of the metrics to compute similarity.

However, for our zero distance constraint it is guaranteed that using the features instead of metrics we would have obtained the same results.

[1] We are using as our development environment the commercial tool REFINE (a trademark of Reasoning Systems Corp.)

[2] Variables are also referred in the text as Identifiers

Features	*Correlation Value*
Fanout & Globals	0.65
Fanout & Globals-Updated	0.59
Fanout & Params-Updated	0.53
Fanout & Read-Input	0.5
Fanout & Files-Opened	0.5
Fanout & Formal-Params	0.05
Fanout & Ids-Used	0.6
Fanout & Fcn-Calls-To-Construct	0.3
Globals & Globals-Updated	0.58
Globals & Params-Updated	0.54
Globals & Read-Input	0.5
Globals & Files-Opened	0.502
Globals & Formal-Params	0.26
Globals & Ids-Used	0.62
Globals & Fcn-Calls-To-Construct	0.29
Globals-Updated & Params-Updated	0.54
Globals-Updated & Read-Input	0.56
Globals-Updated & Files-Opened	0.56
Globals-Updated & Formal-Params	0.29
Globals-Updated & Ids-Used	0.57
Globals-Updated & Fcn-Calls-To-Construct	0.31
Params-Updated & Read-Input	0.82
Params-Updated & Files-Opened	0.82
Params-Updated & Formal-Params	0.41
Params-Updated & Ids-Used	0.46
Params-Updated & Fcn-Calls-To-Construct	0.36
Read-Input & Files-Opened	1.0
Read-Input & Formal-Params	0.5
Read-Input & Ids-Used	0.34
Read-Input & Fcn-Calls-To-Construct	0.42
Files-Opened & Formal-Params	0.5
Files-Opened & Ids-Used	0.39
Files-Opened & Fcn-Calls-To-Construct	0.42
Formal-Params & Ids-Used	0.56
Formal-Params & Fcn-Calls-To-Construct	0.31
Ids-Used & Fcn-Calls-To-Construct	0.29

Table 1: The correlation values for the features used.

Let's consider now the metrics that are calculated using the features above. We have considered five well known metrics, namely:

- S-Complexity

  Description : S_COMPL(a_constr) is equal to

  $$|FAN\_OUT(a\_constr)|^2$$

  where $|FAN\_OUT(a\_constr)|$ is the number of individual function calls in the construct (a_constr)

- D-Complexity

  Description : D_COMPLEXITY(a_constr) is equal to

  $$\frac{|GLOBALS(a\_constr)|}{(|FAN\_OUT(a\_constr)|+1)}$$

  where $|GLOBALS(a\_constr)|$ is the number of individual declarations of global variables used or updated within the construct *a_constr*. A global variable for a Statement or Expression or Function is a variable which is not declared in the Statement, the Expression or the Function.

- McCabe Complexity

  Description : MCCABE(a_constr) is equal to

  $$\epsilon - n + 2$$

  where ϵ is the number of edges in the control flow graph of the construct *a_constr* and n is the number of nodes in the same graph.
  Alternatively McCabe metric can be calculated as

  $$MCCABE(constr) = 1 + d$$

  where d is the number of control decision predicates in the construct *a_constr*

- Albrecht Metric

  Description : ALBRECHT(a_constr) complexity is equal to

  $$\begin{cases} p_1 * |GLOBALS(a\_constr)| + \\ p_2 * (|GLOBALS\_UPDATED(a\_constr)| + \\ |PARMS\_BY\_REF\_UPDATED(a\_constr)|) + \\ p_3 * |READ\_STATS(a\_constr)| + \\ p_4 * FILES\_OPENED(a\_constr) \end{cases}$$

  where

  - $|GLOBALS(a\_constr)|$ is the is the number of individual declarations of global variables used or updated within the construct *a_constr.*

  - $|GLOBALS\_UPDATED(a\_constr)|$ is the number of individual declarations of global variables updated within the construct *a_constr.*

  - $|PARMS\_BY\_REF\_UPDATED(a\_constr)|$ is the number of pointer type variables in the formal parameter list of the Function in which a_constr is contained and which variables are updated within the construct *a_constr* [3].

  - $|READ\_STATS(a\_constr)|$ is the number of input statements in the construct *a_constr.* These statements include the C statements : sscanf, scanf, fscanf, getc, getchar, gets, fgetc, and fgets.

  - FILES_OPENED(a_constr) is the number of fopen statements in the construct a_constr.

  - The parameters p_i have integer values. The current implementation uses the following values[Adamov87] : $p_1 = 4$, $p_2 = 5$, $p_3 = 4$ and, $p_4 = 7$.

[3]Updates are calculated based on Assignment, Pre/Post Incrementation, and Pre/Post Decrementation statements in C programs

- Kafura Metric

  Description : KAFURA(a_constr) complexity is equal to

$$\left\{ \begin{array}{l} (KAFURA\_IN(a\_constr)* \\ KAFURA\_OUT(a\_constr))^2 \end{array} \right.$$

  where

  1. KAFURA_IN(a_constr) is the sum of
     (a) the number of formal parameters ($|FORMAL\_PARMS(a\_constr)|$)
     (b) the number of variables ($|IDS\_USED(a\_constr)|$) used in the construct *a_constr*,
     (c) the number of Function Calls to *a_constr* ($|FUNCTION\_CALLS\_TO(a\_constr)|$)
  2. KAFURA_OUT(a_constr) is the sum of
     (a) number of Functions called by *a_constr* (that is the same as $|FAN\_OUT(a\_constr)|$),
     (b) the number of individual declarations of global variables updated within the construct *a_constr* (that is $|GLOBALS\_UPDATED(a\_constr)|$),
     (c) the number of pointer type variables in the formal parameter list of the function in which *a_constr* is contained and which variables are updated within the construct *a_constr*, that is $|PARMS\_BY\_REF\_UPDATED(a\_constr)|$.

These metrics provide a 5-element tuple that can be used as a signature for a particular code fragment. A more detailed description of these metrics can be found in [Buss94], [Adamov87].

As the metrics are computed compositionally on the AST, each AST node (expression,statement, function) has a metric signature as annotation. Once a code fragment is chosen, then its signature is retrieved and matched against all nodes of the AST (i.e. expression, statement, function) that have the same signature. Heuristics can be used to speed-up the matching process (i.e. do not attempt to match an expression with a function).

In the following sections we adopt an information retrieval approach for investigating the use of the proposed signatures for the identification of programming patterns.

3 Information Retrieval Framework

In this section we lay the framework within which our experiments were conducted.

Software System	LOC	# of Files	# of Functions
TCSH	44,754	46	658
CLIPS	32,807	40	705
BASH	27393	63	632
ROGER	13,615	39	235

Table 2: The Software Systems Used for Experimentation

The software systems used for evaluating the proposed approach are illustrated in Table.2. *Clips* is an expert system shell, developed at NASA's Software Technology Center, *tcsh*, and *bash* are popular Unix shells and *Roger* is a real-time speech recognition system developed at McGill University.

An important point for computing *Recall* and *Precision* is the definition of a measure of *relevance* between a pattern and a retrieved code fragment. To our knowledge, there is no formal definition of *relevance* between two code fragments and there are no standard criteria to recognize one code fragment as being a clone of another. In relevant research studies [Baker95], [Johnson96], [Halst77], [McCabe90], [Jankowitz88], code cloning has been seen as a problem of examining statistical, or textual properties of the code. However, experts make fine distinctions on the operations and the criteria for code cloning. Programmers may argue that textual similarity is the most important criterion. Others may argue that the semantics of the system and the Input/Output relations are more important. Within this framework a safe assumption is to: *a)* use the definitions of code cloning appearing in the literature [Baker95] and, *b)* obtain feed-back from programmers on establishing the relevant data set for each query. Results obtained for each query are tested against this set to establish Recall, and Precision measurements.

For this paper we consider four different cloning scenaria between two code fragments C_1, C_2:

1. C_2 is a clone of C_1, if C_1 and C_2 are syntactically identical (e.g. are found identical using the Unix utility `diff`). That is, there is an identity function F_i on each non blank and printable character in the program text such that $F_i(C_1) = C_2$. In this context equality means syntactic equality.

2. C_2 is a clone of C_1, if C_1 and C_2 have the same structure but modified variable names or data types. That is, there is a substitution θ, of variables and data types used or defined in a code fragment such that, $C_1 = C_2|\theta$. Similarly, in this context equality means syntactic equality.

3. C_2 is a clone of C_1, if C_1 and C_2 have the same structure but modified statements or expressions. That is, there is a substitution σ of statements or expressions such that $C_1 = C_2|\sigma$. Similarly, in this context equality means syntactic equality.

4. C_2 is a clone of C_1, if one differs from the other on inserted, deleted or substituted statements and expressions. That is there is a function $F_{dp}(C_1, I, D)$ and a minimal [4] set of insertions and deletions, I, D respectively, of statements or expressions on C_1 such that by applying F_{dp} we obtain, $F_{dp}(C_1, I, D) = C_2$. Note that in this context equality means syntactic equality.

These scenaria can be combined (using functional composition), and comply with the text-based and measurement-based approaches found in the relevant literature [Mayrand96], [Baker95], [Johnson96] and, [Paul94]. Specifically, in [Baker95] and [Johnson96] exact duplication and parameterized duplication is covered in scenaria one and two above. In [Mayrand96], a number of criteria are proposed. The *exact copy name* criteria are subsumed by our first and second scenario respectively. The *control flow* criterion is subsumed by our second scenario. Finally, the *layout* and *expression* criteria are subsumed by our third and fourth scenario respectively. In [Paul94], a regular language is proposed to identify programming patterns. Cloning can be detected using this system based on the assumption that if two code fragments can be generated by the same pattern then they could be clones. The criteria in [Paul94] for cloning, are subsumed by our second, third and, fourth Scenario. In particular, in [Paul94] wild characters correspond to arbitrary insertions and identifier place-holders correspond to substitutions (θ, σ).

As far as the semantic-based approaches are concerned, we believe that these can be covered mostly in the framework of language semantics and other formal techniques that can be used to indicate functional or behavioral similarity between code fragments [Stoy77]. Note that in general, functional and behavioral equivalence is an undecidable problem, and even for the relaxed conditions where we may prove behavioral similarity most of these techniques are not tractable [Quilici96]. The ultimate goal of IR is to retrieve components to be presented to a user who makes the final decision on their appropriateness. Furthermore, we feel that the semantics approach exceeds the scope of the pattern-matching based framework proposed in this paper.

[4]The size of maximum number of allowable insertions and deletions is defined by the Software Engineer. The more insertions and deletions are allowed the less similar one software element is to another [Wu92].

Following standard Information Retrieval (IR) practice, consultation with programmers of the subject systems was performed in advance to select code fragments that were replicated in the system. Each replicated component has been tagged by its location (file, length in lines of code) and content (number and type of statements it contains). These "tagged" components form the basis for evaluating query results and thus calculating Recall and Precision.

We have considered a space of 940 functions and we formulated 20 queries in total for each method considered. This ratio corresponds well with the same number of queries per number of documents that has been used in standard Information Retrieval test sets [Maarek91], with reported ratios in the range of 2.3% to 2.6% have been reported. The queries were selected by programmers that have experience with the structure and the contents of the subject systems. Essentially, these queries are code fragments for which the programmers knew to be or, to have cloned instances in the subject system. Code fragments were selected based on:

- The knowledge that these were replicated components,
- Coverage of the cloning scenaria discussed above.

Obtained results were checked against the set of the relevant components that have been identified by the programmers.

Let C be a collection of software components in a repository (i.e. all functions in an application). For each query Q, the set C can be partitioned into two disjoint sets: A containing R relevant to the query elements and A' containing irrelevant to the query material.

Now suppose we apply the query using a matching mechanism. In an ideal scenario we would have retrieved R components. Suppose that, for each such query we retrieve a set of c components. Suppose this set contains r relevant to the query components ($r \preceq c$ and ideally $r = R$). Following the classical Information Retrieval (IR) terminology *Recall* and *Precision* are defined as:

$$Recall = \frac{r}{R}$$

$$Precision = \frac{r}{c}$$

In other words, Recall provides the percentage of elements retrieved, measured against the relevant elements that exist in the repository and could have been retrieved with a perfect and ideal matcher. Ideally Recall is 100%.

Precision measures the noise we obtain in our results (i.e. irrelevant documents appearing in the results of a query).

The relationship between Recall and Precision shows how well a matching engine performs. In an ideal matcher, the Precision will remain high as the Recall increases. However, this does not happen in practical applications. The more the constraints of a query are relaxed to retrieve more relevant components (i.e. increase Recall), the more noise is presented in the results (i.e. Precision decreases).

In this study, the relationship between Recall and Precision has been computed using the standard IR approach which consists of:

- Evaluating Recall and Precision for each query at given cut-off points,
- Performing macro-averaging [Jones81] so as to obtain a single Recall and Precision value for every given cut-off point,
- Using linear interpolation to obtain Precision values for Recall values that were not effectively achieved.

Linear interpolation was used to compute Precision values p^* for standard Recall values r^* by applying the following formula :

$$p^* = p_1 + \frac{r^* - r_1}{r_2 - r_1}(p_2 - p_1)$$

where r_1, and r_2 are the recall values immediately to the left and to the right of r^* and p_1, p_2 are the corresponding precision values.

4 Pattern Matching Experiments

In this paper we present the results of clone detection using the metrics described above. Alternatively, we could have used the features instead of the metrics as several metrics use common features. However, we consider in our experiments results obtained with zero distance, and therefore is irrelevant if we use the metrics or the features instead. In order to do that one must first define a similarity ordering relation between software elements. Such an ordering relation could specify when two components are more similar as opposed to the comparison with other components.

Note that, for the interpretation of the results, the reader should not consider how the features are used (quadratically or reciprocally) but should focus on the fact, the features should perfectly match to give zero distance results and therefore the significant factor is that they are used in the calculation of the metric value. In this case, if the metrics have distance zero, then the features should also match exactly.

4.1 Precision Per Metric Usage at Max. Recall Level

This experiment illustrates the Precision [5] variation per metric combination used [6], at Recall level 95.8% (which was the highest effectively achieved Recall level in these experiments), and similarity threshold distance 0.0[7].

- Using One Metric: among the single metric usage scenario the Precision is higher when using the Kafura metric (Fig.1). An explanation for this behavior is the complexity of the metric in terms of the variety of the features used to compute it. Note that in percentage values the Precision is illustrated in Fig.1 where, the Kafura metric achieves a 1.7% Precision value for 95.8% Recall.
- Using Two Metrics: the Precision increases when using the combination of the *Kafura* metric with *S-Complexity* (Fig.2). An explanation is the reduction in noise introduced by intersecting the *FANOUT* feature in both metrics. However, it is interesting to observe how the McCabe metric contributes to the second best Precision value when combined with the *Kafura* metric. The reason for this is that McCabe introduces to the matching process a new low correlated feature (estimated as 0.288), and namely, the structure of the Control Flow Graph. At this point it is useful to distinguish between the two types of influence when two or more metrics are combined in the matching process. We refer to the first type

[5] Please note that the Precision values illustrated in the figures are given in percentage points

[6] A-K denotes the combination of Albrecht and Kafura metric, and D-S the combination of D-Complexity and S-Complexity metrics

[7] Distances between code signatures were calculated using the Euclidean distance in the metric space

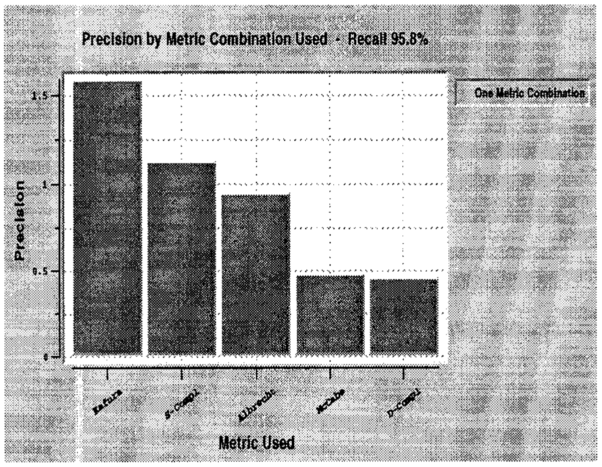

Figure 1: Precision values (in percentage points) for one Metric used (Recall level 95.8%.)

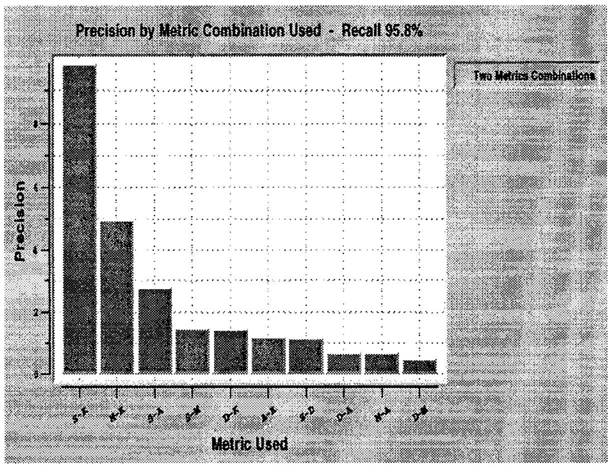

Figure 2: Precision values (in percentage points) for combinations of two Metrics (Recall level 95.8%.)

of influence as *feature intersection. Feature intersection* enhances the Precision because it imposes a common matching criterion. In other words the resulting candidate components are constrained to have similar values in one or more of the features. We refer to the second type of influence as *feature addition. Feature addition* enhances the Precision because it imposes more features in the matching process to be considered. The experiments indicate that a combination of types of influence is better than considering *feature addition* alone (see S-K, M-K combinations in Fig.2, and S-A-K, S-M-A in Fig.3). That means by just adding new features in the matching process does not always result in significant increase in Precision.

- Using Three Metrics: the Precision increases when using a combination of the *Kafura* metric, the *McCabe* metric and, *S-Complexity* (Fig. refm-e3). This result is expected as it includes the metrics from the first best two metrics combinations. The interesting point though is the high Precision we obtain at high Recall levels by the use of the *D-Complexity* or *Albrecht* instead of the *McCabe* metric. This result can be explained by the nature of the *D-Complexity* metric, which essentially imposes a *feature intersection* influence, (*GLOBALS*) with the *Kafura* metric. The same can be said for the *Albrecht* metrics, which imposes a *feature intersection* influence on the *PARAMS_BY_REF_UPDATED* and *GLOBALS_UPDATED*.

- Using Four Metrics: the Precision increases when using a combination of the *Kafura* metric, the *McCabe* metric, the *S-Complexity* metric , and the *D-Complexity* metric (Fig.4). The interesting point here is the possibility of replacing the *D-Complexity* with the *Albrecht* metric. This replacement can be explained by the *feature addition* influence of the I/O features (*READ_STATS*, *FILES_OPEN*) of the *Albrecht* metric. Note that *globals* and *FANOUT* in *D-Complexity* have already been covered by the *S-Complexity* and *Kafura* metric. At this point the *Albrecht* metric adds new matching features and this is the reason it makes such a high contribution.

- Using Five Metrics: we can achieve an effective Recall level of 95.8% and maintain a Precision level of 10.2% (Fig.5). This is not a discouraging

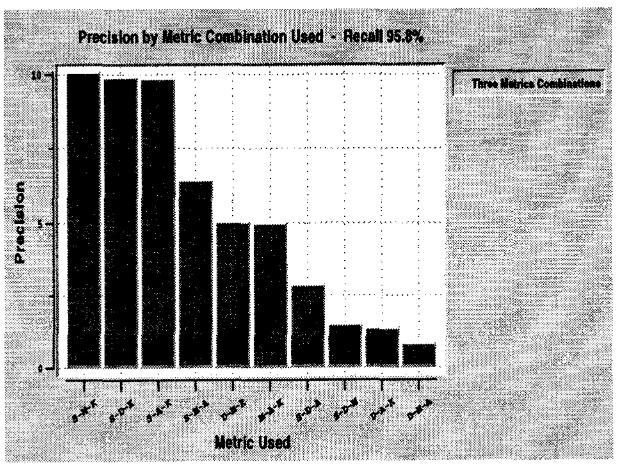

Figure 3: Precision values (in percentage points) for combinations of three Metrics (Recall level 95.8%.)

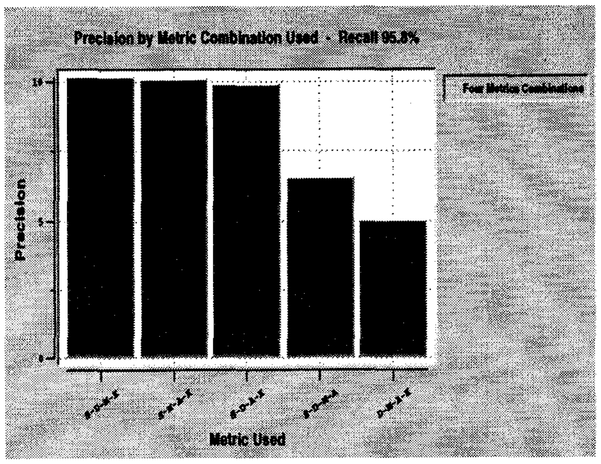

Figure 4: Precision values (in percentage points) for combinations of four Metrics (Recall level 95.8%.)

result if one considers that in all of our experiments (involving the samples queries, and brute-force comparison between all function pairs)[8] we did not retrieve more that 11.3% of the total system size. That means using this approach we will retrieve 11.3% of the system for which we know there are 95.8% of the existing clones, for a given pattern.

4.2 Recall Per Distance Range

This experiment illustrates the significance of the distance values to the recognition process. Using all five metrics with threshold set to 2.5 units[9] and cut-off values illustrated in Table.3 we measured the recall using our sample queries. The result drawn from this experiment is that using the metrics-based pattern matching technique we obtain most of the clones (57.7%), at distance values $\preceq 0.6$. This result indicates that this technique can be used as a fast first approximation to the clone detection problem, for distances close to 0.0. Note that on distances close to 0.0 and using all five metrics this technique is the simplest and fastest to apply when compared on its generality for approximate matching with the other string-based techniques. Moreover, the only additional computation involved in this technique is the comparison of the metric values as the metrics are calculated right after parsing, at link time. Note that the Recall/Precision graph illustrated in the following section suggests for such Recall level a Precision of 46.5% which is a good indication for the usefulness of this technique. The drawback of this technique is that it is language dependent as it requires metrics to be computed first.

4.3 Recall / Precision

Precision values for specific Recall values were computed by performing an Information Retrieval Experiment. Average Recall values and average Precision values have been used to produce, with linear interpolation, Precision values for standard Recall values (0.0, 0.1, 0.2, .. 1.0). Because of the erratic nature of low recall values for small samples [Jones81] we assign ed a Precision value of 1.0 to Recall value of 0.0. We formulated 20 queries for which programmers, have

[8]For the Clips system we had 248,160 possible pair comparisons.

[9]Using the Euclidean distance. Note that, if other distance metrics had been used, different values in Table.3 would have been obtained. However, this table presents the general behavior of a metrics-based cloning detection system.

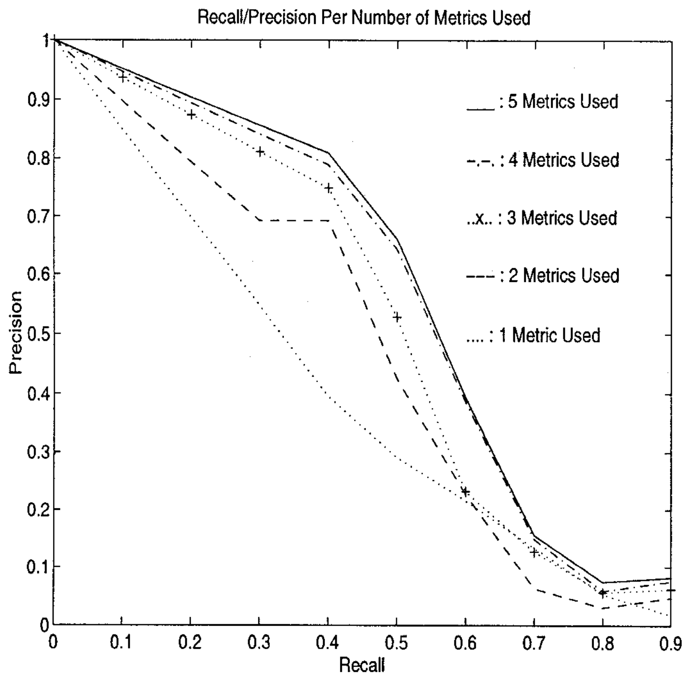

Figure 5: Precision/Recall Graph for different metric combinations. The metric combinations were selected among the ones that give the highest precision in their category class (i.e. the best combination of two metrics is S-Complexity and Kafura)

identified replicated components in the source code.

In this experiment we report the relation of linear Recall to average Precision, using the best combinations obtained by the experiment discussed in Section 4.1.

The obtained results indicate that:

- Using one Metric (Kafura): [10]. we have an almost linear drop in Precision and we obtain a low Precision for high Recall values. The matching time using this method is acceptable but was 64% higher than the best time performance we can obtain. The drawback of using only one metric is the drop in Precision which is almost 7 times lower than the best we can achieve for the highest Recall value.
- Using two Metrics (Kafura, S-Complexity): we have a significant gain in Precision for low to medium Recall levels. This can be explained by the common constraint imposed by the *FANOUT* feature in both metrics. At higher Recall levels we achieved lower values than the ones achieved using the Kafura metric alone. We suspect, though, this is because of the interpolation noise in previous curve above (one metric).
- Using three Metrics (Kafura, S-Complexity, McCabe): we have a new gain in Precision for corresponding Recall values. At this point the McCabe metric is the factor for the Precision increase as it adds the Control Flow component to the already considered features.
- Using Four Metrics(Kafura, D-Complexity, McCabe, S-Complexity): we have a new gain in Precision that can be explained by the common constraints introduced by the common features used for *D-Complexity*, *S-Complexity*, and the *Kafura* metric, combined with a new feature introduced by the *McCabe* metric.
- Using Five Metrics: we obtain the best curve. This is the best combination to use among the metrics we considered. However, it is very close to the one using four metrics. This may suggest dropping the *Albrecht* metric altogether or replace it with *S-Complexity*. This would be a reasonable idea, but note that the metrics are computed at link time at linear complexity on the AST nodes, and therefore do not constitute a significant computation bottleneck. In any case, each metric has its advantages and the choice can be based on the specific goal (speed over accuracy) . D-Complexity is easier and little bit faster to compute, but Albrecht is sensitive to more features and it is better to use in the long run. Experiments at higher levels of granularity (function level, also shown in Table. 4) indicate that

[10]Please refer to the *Kafura* selection in Fig.1

Distance Range	*Recall*
0.0	44.3%
$\preceq$ 0.2	55.4%
$\preceq$ 0.4	57.7%
$\preceq$ 0.6	57.7%
$\preceq$ 0.8	68.2%
$\preceq$ 1.0	68.2%
$\preceq$ 1.2	82.1%
$\preceq$ 1.4	82.1%
$\preceq$ 1.6	82.1%
$\preceq$ 1.8	82.1%
$\preceq$ 2.5	96.0%

Table 3: Recall / Distance Value Range (Metrics)

Metric Combination	Potential Clone Pairs Retrieved	Retrieval Time (Hr:Min:Sec)
K	1,777	0:14:05
A	14,953	0:42:20
D	15,023	0:42:33
S	17,100	0:54:45
M	26,526	1:40:53
D-K	529	0:09:07
A-K	534	0:11:08
M-K	608	0:12:33
S-K	863	0:11:10
S-D	1,841	0:10:23
S-D	2,365	0:12:57
S-M	3,935	0:12:40
D-M	4.032	0:11:55
M-A	4,100	0:13:30
D-A	12,381	0:17:44
D-M-K	283	0:08:24
M-A-K	288	0:10:47
S-A-K	319	0:08:56
S-D-K	321	0:09:27
S-M-K	339	0:10:00
D-A-K	523	0:10:14
S-M-A	942	0:11:14
S-D-M	1,034	0:10:29
S-D-A	1,837	0:12:01
D-M-A	3,677	0:11:45
S-M-A-K	231	0:09:31
S-D-M-K	231	0:09:31
D-M-A-K	282	0:09:29
S-D-A-K	319	0:10:12
S-D-M-A	942	0:10:15
S-D-M-A-K	231	0:05:27

Table 4: Metrics-based matching statistics. The size of all possible pairs for this experiment is 248,160. The Recall level for this experiment using all five metrics is estimated as 44.4%.

when *D-Complexity* and *Albrecht* metrics are used alone, then the *Albrecht* metric generates fewer candidates, but when combined with other metrics this trend does not always hold (D-K and A-K, D-M-K and M-A-K, S-M-A-K and S-D-M-K)[11].

The time statistics for the proposed approach is illustrated in Table.4

The time statistics in Table.4 indicate that when *D-Complexity*, *McCabe Complexity* and the *Kafura* metric are combined give the highest performance to effort ratio (i.e. only three metrics are used and we get only 19% noise on the candidates we get when all of our metrics used in the experiment are considered).

Finally, the cloning detection framework presented above can be modified so that the features are used instead of metrics. However, the results for the experiments presented above would be identical as if features have been used instead of metrics because the presented framework assumes zero distance values (i.e. all features, and metrics match exactly). If other similarity criteria were adopted (as in [Mayrand96]) then would make sense to consider individual features (sensitive to the criterion tested) as opposed to combined metrics. However, similarity criteria, do not imply that he have achieved a universal way to define what a clone really is. Merely, such similarity criteria set a subjective set of axioms for experimentation.

[11] Please refer to Table.4

5 Conclusion

We have experimented with a number of program features that are used to compute five standard software engineering metrics that classify and represent a code fragment. These metrics can be computed compositionally by using the Abstract Syntax Tree at link time. The matching technique discussed in this paper is based on the assumption that if two code fragments are clones, then they share a number of structural and data flow characteristics that can be effectively classified by these metrics. Our experiments show that the metrics-based approach provides a fast approximation of the code cloning recognition problem when partial and approximate matching are essential requirements. Experimental results have indicated that we can effectively retrieve 60% of the code cloning instances sought, and maintain a Precision of approximately 41.0% at the final results. The strength of this approach is that it can be easily used, does not depend on any complex formalism to represent source code entities, and it is time and space efficient, as it is mostly based on comparison of numeric tuples. We believe we can speed-up the performance of the matching algorithm by introducing heuristics such as computation of "hash" values from the five metrics and use this to reduce the set of candidates to be fed to more elaborate and accurate distance measurement algorithms (i.e. Euclidean distance). The price to pay for the speed and ease of use of this method is that at higher Recall levels noise can be introduced and low Precision values be obtained. At a Recall level of 70.0% the Precision can drop to 19.2%. However, this is not problematic as only a small fraction of the system is retrieved (in our experiments $\preceq$ 11.3% of the total size of the system) and therefore can be used at a pre-processing stage to limit the search space when using more accurate but more computationally expensive methods.

References

[Adamov87] Adamov, R. "Literature review on software metrics", *Zurich: Institut fur Informatik der Universitat Zurich*, 1987.

[Baker95] Baker S. B, "On Finding Duplication and Near-Duplication in Large Software Systems" *In Proceedings of the Working Conference on Reverse Engineering 1995*, Toronto ON. July 1995i, pp. 86-95.

[Bowdidge96] Bowdidge, R., "Star diagrams: Designing abstractions out of existing code", *OOPSLA'96 Workshop on Transforming Legacy Systems to Object Oriented Systems*, San Jose Ca., 1996.

[Buss94] E. Buss, R. De Mori, W. M. Gentleman, J. Henshaw, H. Johnson, K. Kontogiannis, E. Merlo, H. A. Müller, J. Mylopoulos, S. Paul, A. Prakash, M. Stanley, S. R. Tilley, J. Troster and K. Wong, "Investigating Reverse Engineering Technologies for the CAS Program Understanding Project", IBM Systems Journal, vol. 33 no. 3, 1994, pp. 477-500.

[Canfora94] Canfora, G., Cimitile, A., DeLucca, A., "Software Salvaging Based on Conditions" *IEEE Conf. on Software Maintenance*, 1994, pp. 424-433.

[Halst77] Halstead, M., H., "Elements of Software Science", New York: Elsevier North-Holland, 1977.

[Hanau80] Hanau, R. and Lenorovitch, R., "Prototyping and Simulation Tools for User/Computer Dialogue Design," *Proceedings of the ACM SIGRAPH 80 7th Annual Conference on Computer Graphics and Interactive Techniques*, Seattle Wash., 1980.

[Jankowitz88] Jankowitz, H., T., "Detecting Plagiarism in student PASCAL programs" *Computer Journal*, 31(1):1-8, 1988.

[Johnson96] Johnson, H., "Navigating the Textual Redundancy Web in legacy Source" *In Proceedings, CASCON'96*, Toronto, ON, Nov. 12-14, 1996, pp.7-16

[Jones81] Jones, K., "Information Retrieval Experiment" Butterworths Publishing Co., Toronto, 1981.

[Konto96] Kontogiannis K., DeMori, R., Merlo, E., Galler, M., Bernstein, M., "Pattern Matching for Clone and Concept Detection", Journal of Automated Software Engineering, vol.3, 1996, pp.77-108.

[Maarek91] Maarek Y., Berry, D., Kaiser, G., "An Information Retrieval Approach For Automatically Constructing Software Libraries", *IEEE Transactions in Software Engineering*, vol.17, No. 8, August 1991, pp.800-813.

[Mayrand96] Mayrand, J., Leblanc, C., Merlo, E., "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics", *International Conference on Software Maintenance 1996*, Monterey Ca., pp.244-253.

[McCabe90] McCabe T., J., "Reverse Engineering, reusability, redundancy: the connection" *American Programmer*, 3(10):8-13, Oct. 1990.

[Muller93] Müller, H.A., "Understanding Software Systems Using Reverse Engineering Technology Perspectives from the Rigi Project" *In Proceedings of CASCON'93*, Toronto, ON. 24-28 Oct. pp. 217-226.

[Muth] Muth, R.,Manber U.,"Approximate Multiple String Matching",

http://www.glimpse.cs.arizona.edu/udi.html

[Myers89] Myers, E., Miller W. "Approximate Matching of Regular Expressions", *Bulletin of Mathematical Biology*, 51(1), 1989, pp.5-37.

[Newcomb97] Newcomb, P., P. Scott, Requirements for Advanced Year 2000 Maintenance Tools", *IEEE Computer*, March 1997, pp.52-57.

[Ning94] Ning, J., Engberts, A., Kozaczynski, W., "Automated Support for Legacy Code Understanding", *Communications of the ACM*, May 1994, Vol.37, No.5, pp.50-57.

[Paul94] Paul, S., Prakash, A., "A Framework for Source Code Search Using Program Patterns", *IEEE Transactions on Software Engineering*, June 1994, Vol. 20, No.6, pp. 463-475.

[Quilici96] Quilici "Reengineering of Legacy Systems: Is it Doomed to Failure?", *International Conference on Software Engineering*, Berlin 1996

[Rugaber96] Rugaber,S., Stirewalt, K., Wills, L., "Understanding Interleaved Code", *Automated Software Engineering*, vol. 3, pp. 47-76, 1996.

[Stoy77] Stoy, J.E., *Denotational Semantics*, MIT Press, 1977.

[Wills93] Wills, L.M., "Automated Program Recognition by Graph Parsing" *MIT Technical Report 1358*, MIT, AI Laboratory, 1993

[Wong95] Wong, K., Tilley, S., Müller, H., Storey, M-A., "Structural re-documentation: A case study", *IEEE Software*, 12(1), January 1995, pp. 46-54.

[Wu92] Wu, S., Manber, U., "Agrep - A fast approximate pattern matching tool", *Usenix Winter 92 Technical Conference*, San Fransisco, Ca., January 1992, pp. 153-162.