

Pattern Matching for Clone and Concept Detection *

K. A. KONTOGIANNIS, R. DEMORI, E. MERLO, M. GALLER, M. BERNSTEIN

kostas@cs.mcgill.ca

McGill University School of Computer Science 3480 University St., Room 318, Montréal, Canada H3A 2A7

Abstract.

A legacy system is an operational, large-scale software system that is maintained beyond its first generation of programmers. It typically represents a massive economic investment and is critical to the mission of the organization it serves. As such systems age, they become increasingly complex and brittle, and hence harder to maintain. They also become even more critical to the survival of their organization because the business rules encoded within the system are seldom documented elsewhere.

Our research is concerned with developing a suite of tools to aid the maintainers of legacy systems in recovering the knowledge embodied within the system. The activities, known collectively as “program understanding”, are essential preludes for several key processes, including maintenance and design recovery for reengineering.

In this paper we present three pattern-matching techniques: source code metrics, a dynamic programming algorithm for finding the best alignment between two code fragments, and a statistical matching algorithm between abstract code descriptions represented in an abstract language and actual source code. The methods are applied to detect instances of code cloning in several moderately-sized production systems including tcsh, bash, and CLIPS.

The programmer’s skill and experience are essential elements of our approach. Selection of particular tools and analysis methods depends on the needs of the particular task to be accomplished. Integration of the tools provides opportunities for synergy, allowing the programmer to select the most appropriate tool for a given task.

Keywords: reverse engineering, pattern matching, program understanding, software metrics, dynamic programming

1. Introduction

Large-scale production software systems are expensive to build and, over their useful lifetimes, are even more expensive to maintain. Successful large-scale systems are often called “legacy systems” because (a) they tend to have been in service for many years, (b) the original developers, in the normal course of events, move on to other projects, leaving the system to be maintained by successive generations of maintenance programmers, and (c) the systems themselves represent enormous corporate assets that cannot be easily replaced.

Legacy systems are intrinsically difficult to maintain because of their sheer bulk and because of the loss of historical information: design documentation is seldom maintained as the system evolves. In many cases, the source code becomes the sole repository for evolving corporate business rules.

* This work is in part supported by IBM Canada Ltd., Institute for Robotics and Intelligent Systems, a Canadian Network of Centers of Excellence and, the Natural Sciences and Engineering Research Council of Canada. Based on “Pattern Matching for Design Concept Localization” by K.A.Kontogiannis, R.DeMori, M.Bernstein, M.Galler, E.Merlo, which first appeared in Proceedings of the Second Working Conference on Reverse Engineering, pp.96-103, July, 1995, © IEEE, 1995

During system maintenance, it is often necessary to move from low, implementation-oriented levels of abstraction back to the design and even the requirements levels. The process is generally known as “reverse engineering”.¹ In (Chikofsky, 1990) there are definitions for a variety of subtasks, including “reengineering”, “restructuring”, and “redocumentation”.

In particular, it has been estimated that 50 to 90 percent of the maintenance programmer’s effort is devoted to simply understanding relationships within the program. The average Fortune 100 company maintains 35 million lines of source code (MLOC) with a growth rate of 10 percent per year just in enhancements, updates, and normal maintenance. Facilitating the program understanding process can yield significant economic savings.

We believe that maintaining a large legacy software system is an inherently human activity that requires knowledge, experience, taste, judgement and creativity. For the foreseeable future, no single tool or technique will replace the maintenance programmer nor even satisfy all of the programmer’s needs. Evolving real-world systems requires pragmatism and flexibility.

Our approach is to provide a suite of complementary tools from which the programmer can select the most appropriate one for the specific task at hand. An integration framework enables exploitation of synergy by allowing communication among the tools.

Our research is part of a larger joint project with researchers from IBM Centre for Advanced Studies, University of Toronto, and University of Victoria (Buss et al., 1994)

Over the past three years, the team has been developing a toolset, called RevEngE (*Reverse Engineering Environment*), based on an open architecture for integrating heterogeneous tools. The toolset is integrated through a common repository specifically designed to support program understanding (Mylopoulos, 1990). Individual tools in the kit include Ariadne (Konto, 1994), ART (Johnson, 1993), and Rigi (Tilley, 1994). ART (*Analysis of Redundancy in Text*) is a prototype textual redundancy analysis system. Ariadne is a set of pattern matching and design recovery programs implemented using a commercial tool called *The Software Refinery*². Currently we are working on another version of the Ariadne environment implemented in C++. Rigi is a programmable environment for program visualization. The tools communicate through a flexible object server and single global schema implemented using the Telos information modeling language and repository (Mylopoulos, 1990).

In this paper we describe two types of pattern-matching techniques and discuss why pattern matching is an essential tool for program understanding. The first type is based on numerical comparison of selected metric values that characterize and classify source code fragments.

The second type is based on Dynamic Programming techniques that allow for statement-level comparison of feature vectors that characterize source code program statements. Consequently, we apply these techniques to address two types of relevant program understanding problems.

The first one is a comparison between two different program segments to see if one is a clone of the other, that is if the two segments are implementations of the same algorithm. The problem is in theory undecidable, but in practice it is very useful to provide software maintainers with a tool that detects similarities between code segments. Similar

segments are proposed to the software engineer who will make the final decision about their modification or other use.

The second problem is the recognition of program segments that implement a given programming concept. We address this problem by defining a concept description language called ACL and by applying statement-level comparison between feature vectors of the language and feature vectors of source code program statements.

1.1. The Code Cloning Problem

Source code cloning occurs when a developer reuses existing code in a new context by making a copy that is altered to provide new functionality. The practice is widespread among developers and occurs for several reasons: making a modified copy may be simpler than trying to exploit commonality by writing a more general, parameterized function; scheduling pressures may not allow the time required to generalize the code; and efficiency constraints may not admit the extra overhead (real or perceived) of a generalized routine.

In the long run, code cloning can be a costly practice. Firstly, it results in a program that is larger than necessary, increasing the complexity that must be managed by the maintenance programmer and increasing the size of the executable program, requiring larger computers. Secondly, when a modification is required (for example, due to bug fixes, enhancements, or changes in business rules), the change must be propagated to all instances of the clone. Thirdly, often-cloned functionality is a prime candidate for repackaging and generalization for a repository of reusable components which can yield tremendous leverage during development of new applications.

This paper introduces new techniques for detecting instances of source code cloning. Program features based on software metrics are proposed. These features apply to basic program segments like individual statements, `begin-end` blocks and functions. Distances between program segments can be computed based on feature differences. This paper proposes two methods for addressing the code cloning detection problem.

The first is based on direct comparison of metric values that classify a given code fragment. The granularity for selecting and comparing code fragments is at the level of `begin-end` blocks. This method returns clusters of `begin-end` blocks that may be products of cut-and-paste operations.

The second is based on a new Dynamic Programming (DP) technique that is used to calculate the best alignment between two code fragments in terms of *deletions*, *insertions* and, *substitutions*. The granularity for selecting code fragments for comparison is again at the level of `begin-end` blocks. Once two `begin-end` blocks have been selected, they are compared at the statement level. This method returns clusters of `begin-end` blocks that may be products of cut-and-paste operations. The DP approach provides in general, more accurate results (i.e. less false positives) than the one based on direct comparison of metric values at the `begin-end` block level. The reason is that comparison occurs at the statement level and informal information is taken into account (i.e. variable names, literal strings and numbers).

1.2. *The Concept Recognition Problem*

Programming concepts are described by a concept language. A concept to be recognized is a phrase of the concept language. Concept descriptions and source code are parsed. The concept recognition problem becomes the problem of establishing correspondences, as in machine translation, between a parse tree of the concept description language and the parse tree of the code.

A new formalism is proposed to see the problem as a stochastic syntax-directed translation. Translation rules are pairs of rewriting rules and have associated a probability that can be set initially to uniform values for all the possible alternatives.

Matching of concept representations and source code representations involves alignment that is again performed using a dynamic programming algorithm that compares feature vectors of concept descriptions, and source code.

The proposed concept description language, models *insertions* as wild characters (*AbstractStatement** and *AbstractStatement+*) and does not allow any *deletions* from the pattern. The comparison and selection granularity is at the statement level. Comparison of a concept description language statement with a source code statement is achieved by comparing feature vectors (i.e. metrics, variables used, variables defined and keywords).

Given a concept description $\mathcal{M} = A_1; A_2; \dots A_m$, a code fragment $\mathcal{P} = S_1; S_2; \dots S_k$ is selected for comparison if: *a*) the first concept description statement A_1 matches with S_1 , and *b*) the sequence of statements $S_2; \dots S_k$, belong to the innermost begin-end block containing S_1 .

The use of a statistical formalism allows a score (a probability) to be assigned to every match that is attempted. Incomplete or imperfect matching is also possible leaving to the software engineer the final decision on the similar candidates proposed by the matcher.

A way of dynamically updating matching probabilities as new data are observed is also suggested in this paper. Concept-to-code matching is under testing and optimization. It has been implemented using the REFINE environment and supports plan localization in C programs.

1.3. *Related Work*

A number of research teams have developed tools and techniques for localizing specific code patterns.

The UNIX operating system provides numerous tools based on regular expressions both for matching and code replacement. Widely-used tools include `grep`, `awk`, `ed` and `vi`. These tools are very efficient in localizing patterns but do not provide any way for partial and hierarchical matching. Moreover, they do not provide any similarity measure between the pattern and the input string.

Other tools have been developed to browse source code and query software repositories based on structure, permanent relations between code fragments, keywords, and control or dataflow relationships. Such tools include CIA, Microscope, Rigi, SCAN, and REFINE. These tools are efficient on representing and storing in local repositories relationships between program components. Moreover, they provide effective mechanisms for querying

and updating their local repositories. However, they do not provide any other mechanism to localize code fragments except the stored relations. Moreover no partial matching and no similarity measures between a query and a source code entity can be calculated.

Code duplication systems use a variety of methods to localize a code fragment given a model or a pattern. One category of such tools uses structure graphs to identify the “fingerprint” of a program (Jankowitz, 1988). Other tools use metrics to detect code patterns (McCabe, 1990), (Halstead, 1977), common dataflow (Horwitz, 1990), approximate fingerprints from program text files (Johnson, 1993), text comparison enhanced with heuristics for approximate and partial matching (Baker, 1995), and text comparison tools such as `Unix diff`.

The closest tool to the approach discussed in this paper, is SCRUPLE (Paul, 1994). The major improvement of the solution proposed here is *a*) the possibility of performing partial matching with feature vectors, providing similarity measures between a pattern and a matched code fragment, and *b*) the ability to perform hierarchical recognition. In this approach, explicit concepts such as `Iterative-Statement` can be used allowing for multiple matches with a `While`, a `For` or, a `Do` statement in the code. Moreover, recognized patterns can be classified, and stored so that they can be used inside other more complex composite patterns. An expansion process is used for unwrapping the composite pattern into its components.

2. Code to Code Matching

In this section we discuss pattern-matching algorithms applied to the problem of clone detection. Determining whether two arbitrary program functions have identical behavior is known to be undecidable in the general case. Our approach to clone detection exploits the observation that clone instances, by their nature, should have a high degree of structural similarity. We look for identifiable characteristics or features that can be used as a signature to categorize arbitrary pieces of code.

The work presented here uses *feature vectors* to establish similarity measures. Features examined include metric values and specific data- and control-flow properties. The analysis framework uses two approaches:

1. direct comparison of metric values between `begin-end` blocks, and
2. dynamic programming techniques for comparing `begin-end` blocks at a statement-by-statement basis.

Metric-value similarity analysis is based on the assumption that two code fragments C_1 and C_2 have metric values $M(C_1)$ and $M(C_2)$ for some source code metric M . If the two fragments are similar under the set of features measured by M , then the values of $M(C_1)$ and $M(C_2)$ should be proximate.

Program features relevant for clone detection focus on data and control flow program properties. Modifications of five widely used metrics (Adamov, 1987), (Buss et al., 1994) for which their components exhibit low correlation (based on the Spearman-Pierson correlation test) were selected for our analyses:

1. The number of functions called (fanout);

2. The ratio of input/output variables to the fanout;
3. McCabe cyclomatic complexity;
4. Modified Albrecht's function point metric;
5. Modified Henry-Kafura's information flow quality metric.

Detailed descriptions and references for metrics will be given later on in this section. Similarity of two code fragments is measured using the resulting 5-dimensional vector. Two methods of comparing metric values were used. The first, naive approach, is to make $O(n^2)$ pairwise comparisons between code fragments, evaluating the Euclidean distance of each pair. A second, more sophisticated analytical approach was to form clusters by comparing values on one or more axes in the metric space.

The selection of the blocks to be compared is based on the proximity of their metric value similarity in a selected metric axis. Specifically, when the source code is parsed an *Abstract Syntax Tree* (AST) T_c is created, five different metrics are calculated compositionally for every statement, block, function, and file of the program and are stored as annotations in the corresponding nodes of the AST. Once metrics have been calculated and annotations have been added, a reference table is created that contains source code entities sorted by their corresponding metric values. This table is used for selecting the source code entities to be matched based on their metric proximity. The comparison granularity is at the level of a `begin-end` block of length more than n lines long, where n is a parameter provided by the user.

In addition to the direct metric comparison techniques, we use dynamic programming techniques to calculate the best alignment between two code fragments based on *insertion*, *deletion* and *comparison* operations. Rather than working directly with textual representations, source code statements, as opposed to `begin-end` blocks, are abstracted into feature sets that classify the given statement. The features per statement used in the Dynamic Programming approach are:

- Uses of variables, definitions of variables, numerical literals, and strings;
- Uses and definitions of data types;
- The five metrics as discussed previously.

Dynamic programming (DP) techniques detect the best alignment between two code fragments based on *insertion*, *deletion* and *comparison* operations. Two statements match if they *define* and *use* the same variables, strings, and numerical literals. Variations in these features provide a dissimilarity value used to calculate a global dissimilarity measure of more complex and composite constructs such as `begin-end` blocks and functions. The comparison function used to calculate dissimilarity measures is discussed in detail in Section 2.3. Heuristics have been incorporated in the matching process to facilitate variations that may have occurred in cut and paste operations. In particular, the following heuristics are currently considered:

- Adjustments between variable names by considering lexicographical distances;

- Filtering out short and trivial variable names such as `i` and `j` which are typically used for temporary storage of intermediate values, and as loop index values. In the current implementation, only variable names of more than three characters long are considered.

Dynamic programming is a more accurate method than the direct metric comparison based analysis (Fig. 2) because the comparison of the feature vector is performed at the statement level. Code fragments are selected for Dynamic Programming comparison by preselecting potential clone candidates using the direct metric comparison analysis. Within this framework only the `begin-end` blocks that have a dissimilarity measure less than a given threshold are considered for DP comparison. This preselection reduces the comparison space for the more computationally expensive DP match.

The following sections further discuss these approaches and present experimental results from analyzing medium scale ($\leq 100\text{kLOC}$) software systems.

2.1. Program Representation and the Development of the Ariadne Environment

The foundation of the *Ariadne* system is a program representation scheme that allows for the calculation of the feature vectors for every statement, block or function of the source code. We use an object-oriented annotated abstract syntax tree (AST). Nodes of the AST are represented as objects in a LISP-based development environment³.

Creating the annotated AST is a three-step process. First, a grammar and object (domain) model must be written for the programming language of the subject system. The tool vendor has parsers available for such common languages as C and COBOL. Parsers for other languages may be easily constructed or obtained through the user community. The domain model defines object-oriented hierarchies for the AST nodes in which, for example, an *If-Statement* and a *While-Statement* are defined to be subclasses of the *Statement* class.

The second step is to use the parser on the subject system to construct the AST representation of the source code. Some tree annotations, such as linkage information and the call graph are created automatically by the parser. Once the AST is created, further steps operate in an essentially language-independent fashion.

The final step is to add additional annotations into the tree for information on data types, dataflow (dataflow graphs), the results of external analysis, and links to informal information. Such information is typically obtained using dataflow analysis algorithms similar to the ones used within compilers.

For example, consider the following code fragment from an IBM-proprietary PL/1-like language. The corresponding AST representation for the `if` statement is shown in Fig. 1. The tree is annotated with the fan-out attribute which has been determined during an analysis phase following the initial parse.

```
MAIN: PROCEDURE(OPTION);
      DCL OPTION FIXED(31);
      IF (OPTION>0) THEN
          CALL SHOW_MENU(OPTION);
      ELSE
```

```
CALL SHOW_ERROR("Invalid option number");
END MAIN;
```

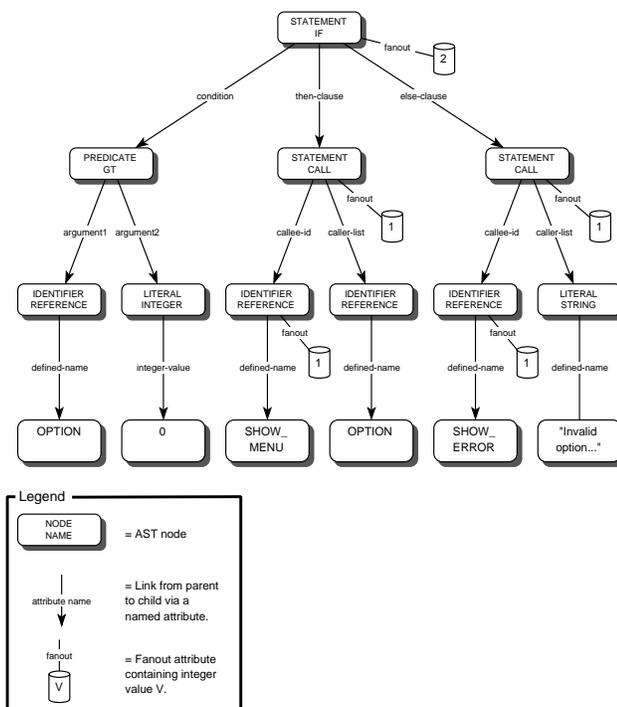


Figure 1. The AST for an IF Statement With Fanout Attributes.

2.2. Metrics Based Similarity Analysis

Metrics based similarity analysis uses five source-code metrics that are sensitive to several different control and data flow program features. Metric values are computed for each statement, block, and function. Empirical analysis⁴ (Buss et al., 1994) shows the metrics components have low correlation, so each metric adds useful information.

The features examined for metric computation include:

- Global and local variables defined or used;

- Functions called;
- Files accessed;
- I/O operations (read, write operations);
- Defined/used parameters passed by reference and by value;
- Control flow graph.

Partial matching may occur because the metrics are not sensitive to variable names, source code white space, and minor modifications such as replacement of `while` with `for` loops and insertion of statements that do not alter the basic data and control flow of the original code structure.

A description of the metrics used is given below but a more detailed description can be found in (Adamov, 1987), (Fenton, 1991), (Moller93).

Let s be a code fragment. The description of the five modified metrics used is given below. Note that these metrics are computed compositionally from statements, to `begin`-end blocks, functions, and files.

1. $S_COMPLEXITY(s) = FAN_OUT(s)^2$

where

- $FAN_OUT(s)$ is the number of individual function calls made within s .

2. $D_COMPLEXITY(s) = GLOBALS(s)/(FAN_OUT(s) + 1)$

where

- $GLOBALS(s)$ is the number of individual declarations of global variables used or updated within s . A global variable is a variable which is not declared in the code fragment s .

3. $MCCABE(s) = \epsilon - n + 2$

where

- ϵ is the number of edges in the control flow graph
- n is the number of nodes in the graph.

Alternatively McCabe metric can be calculated using

- $MCCABE(s) = 1 + d$, where d is the number of control decision predicates in s .

4. $ALBRECHT(s) = \begin{cases} p_1 * VARS_USED_AND_SET(s) + \\ p_2 * GLOBAL_VARS_SET(s) + \\ p_3 * USER_INPUT(s) + \\ p_4 * FILE_INPUT(s) \end{cases}$

where,

- $VAR_USED_AND_SET(s)$ is the number of data elements set and used in the statement s ,
- $GLOBAL_VAR_SET(s)$ is the number of global data elements set in the statement s ,
- $USER_INPUT(s)$ is the number of read operations in statement s ,
- $FILE_INPUT(s)$ is the number of files accessed for reading in s .
- The factors p_1, \dots, p_4 , are weight factors. In (Adamov, 1987) possible values for these factors are given. In the current implementation the values chosen are $p_1 = 5$, $p_2 = 4$, $p_3 = 4$ and, $p_4 = 7$. The selection of values for the p_i 's $\neq 0$ does not affect the matching process.

5. $KAFURA(s) = \{ (KAFURA_IN(s) * KAFURA_OUT(s))^2$ where,

- $KAFURA_IN(s)$ is the sum of local and global incoming dataflow to the the code fragment s .
- $KAFURA_OUT(s)$ is the sum of local and global outgoing dataflow from the the code fragment s .

Once the five metrics M_1 to M_5 are computed for every statement, block and function node, the pattern matching process is fast and efficient. It is simply the comparison of numeric values.

We have experimented with two techniques for calculating similar code fragments in a software system.

The first one is based on pairwise Euclidean distance comparison of all `begin-end` blocks that are of length more than n lines long, where n is a parameter given by the user. In a large software system though there are many `begin-end` blocks and such a pairwise comparison is not possible because of time and space limitations. Instead, we limit the pairwise comparison between only these `begin-end` blocks that for a selected metric axis \mathcal{M}_i their metric values differ in less than a given threshold d_i . In such a way every block is compared only with its close metric neighbors.

The second technique is more efficient and is using clustering per metric axis. The technique starts by creating clusters of potential clones for every metric axis \mathcal{M}_i ($i = 1 \dots 5$). Once the clusters for every axis are created, then intersections of clusters in different axes are calculated forming intermediate results. For example every cluster in the axis \mathcal{M}_i contains potential clones under the criteria implied by this metric. Consequently, every cluster that has been calculated by intersecting clusters in \mathcal{M}_i and \mathcal{M}_j contains potential clones under the criteria implied by *both* metrics. The process ends when all metric axis have been considered. The user may specify at the beginning the order of comparison, and the clustering thresholds for every metric axis. The clone detection algorithm that is using clustering can be summarized as:

1. Select all source code `begin-end` blocks \mathcal{B} from the AST that are more than n lines long. The parameter n can be changed by the user.
2. For every metric axis \mathcal{M}_i ($i = 1 \dots 5$) create clusters $\mathcal{C}_{i,j}$ that contain `begin-end` blocks with distance less than a given threshold d_i that is selected by the user. Each cluster

then contains potential code clone fragments under the metric criterion \mathcal{M}_i . Set the current axis $\mathcal{M}_{curr} = \mathcal{M}_i$, where $i = 1$. Mark \mathcal{M}_i as *used*

3. For every cluster $\mathcal{C}_{curr,m}$ in the current metric axis \mathcal{M}_{curr} , intersect with all clusters $\mathcal{C}_{j,k}$ in one of the non *used* metric axis $\mathcal{M}_j, j \in \{1 .. 5\}$. The clusters in the resulting set contain potential code clone fragments under the criteria \mathcal{M}_{curr} **and** \mathcal{M}_j , and form a composite metric axis $\mathcal{M}_{curr \odot j}$. Mark \mathcal{M}_j as *used* and set the current axis $\mathcal{M}_{curr} = \mathcal{M}_{curr \odot j}$.
4. If all metric axes have been considered the stop; else go to *Step 3*.

The pattern matching engine uses either the computed Euclidean distance or clustering in one or more metric dimensions combined, as a similarity measure between program constructs.

As a refinement, the user may restrict the search to code fragments having minimum size or complexity.

The metric-based clone detection analysis has been applied to a several medium-sized production C programs.

In `tcsh`, a 45 kLOC Unix shell program, our analysis has discovered 39 clusters or groups of similar functions of average size 3 functions per cluster resulting in a total of 17.7 percent of potential system duplication at the function level.

In `bash`, a 40KLOC Unix shell program, the analysis has discovered 25 clusters, of average size 5.84 functions per cluster, resulting in a total of 23 percent of potential code duplication at the function level.

In CLIPS, a 34 kLOC expert system shell, we detected 35 clusters of similar functions of average size 4.28 functions per cluster, resulting in a total of 20 percent of potential system duplication at the function level.

Manual inspection of the above results combined with more detailed Dynamic Programming re-calculation of distances gave some statistical data regarding false positives. These results are given in Table 1. Different programs give different distribution of false alarms, but generally the closest the distance is to 0.0 the more accurate the result is.

The following section, discusses in detail the other code to code matching technique we developed, that is based on Dynamic Programming.

2.3. Dynamic Programming Based Similarity Analysis

The Dynamic Programming pattern matcher is used (Konto, 1994), (Kontogiannis, 1995) to find the best alignment between two code fragments. The distance between the two code fragments is given as a summation of comparison values as well as of insertion and deletion costs corresponding to insertions and deletions that have to be applied in order to achieve the best alignment between these two code fragments.

A program feature vector is used for the comparison of two statements. The features are stored as attribute values in a frame-based structure representing expressions and statements in the AST. The cumulative similarity measure \mathcal{D} between two code fragments P, M , is calculated using the function

$$D : \text{Feature_Vector} \times \text{Feature_Vector} \rightarrow \text{Real}$$

where:

$$D(\mathcal{E}(1, p, \mathcal{P}), \mathcal{E}(1, j, \mathcal{M})) = \text{Min} \begin{cases} \Delta(p, j - 1, \mathcal{P}, \mathcal{M}) + \\ D(\mathcal{E}(1, p, \mathcal{P}), \mathcal{E}(1, j - 1, \mathcal{M})) \\ \\ I(p - 1, j, \mathcal{P}, \mathcal{M}) + \\ D(\mathcal{E}(1, p - 1, \mathcal{P}), \mathcal{E}(1, j, \mathcal{M})) \\ \\ C(p - 1, j - 1, \mathcal{P}, \mathcal{M}) + \\ D(\mathcal{E}(1, p - 1, \mathcal{P}), \mathcal{E}(1, j - 1, \mathcal{M})) \end{cases} \quad (1)$$

and,

- \mathcal{M} is the model code fragment
- \mathcal{P} is the input code fragment to be compared with the model \mathcal{M}
- $\mathcal{E}(i, j, \mathcal{Q})$ is a program feature vector from position i to position j in code fragment \mathcal{Q}
- $D(\mathcal{V}_x, \mathcal{V}_y)$ is the the distance between two feature vectors $\mathcal{V}_x, \mathcal{V}_y$
- $\Delta(i, j, \mathcal{P}, \mathcal{M})$ is the cost of deleting the j th statement of \mathcal{M} , at position i of the fragment \mathcal{P}
- $I(i, j, \mathcal{P}, \mathcal{M})$ the cost of inserting the i th statement of \mathcal{P} at position j of the model \mathcal{M} and
- $C(i, j, \mathcal{P}, \mathcal{M})$ is the cost of comparing the i th statement of the code fragment \mathcal{P} with the j th fragment of the model \mathcal{M} . The comparison cost is calculated by comparing the corresponding feature vectors. Currently, we compare ratios of variables set, used per statement, data types used or set, and comparisons based on metric values

Note that *insertion*, and *deletion* costs are used by the Dynamic Programming algorithm to calculate the best fit between two code fragments. An intuitive interpretation of the best fit using *insertions* and *deletions* is “if we insert statement i of the input at position j of the model then the model and the input have the smallest feature vector difference..”

The quality and the accuracy of the comparison cost is based on the program features selected and the formula used to compare these features. For simplicity in the implementation we have attached constant real values as insertion and deletion costs.

Table 1 summarizes statistical data regarding false alarms when Dynamic Programming comparison was applied to functions that under direct metric comparison have given distance 0.0 . The column labeled *Distance Range* gives the value range of distances between functions using the Dynamic Programming approach. The column labeled *False Alarms* contains the percentage of functions that are not clones but they have been identified as such. The column labeled *Partial Clones* contains the percentage of functions which correspond

Table 1. False alarms for the Clips program

<i>Distance Range</i>	<i>False Alarms</i>	<i>Partial Clones</i>	<i>Positive Clones</i>
0.0	0.0 %	10.0%	90.0%
0.01 - 0.99	6.0 %	16.0 %	78.0%
1.0 - 1.49	8.0%	3.0 %	89.0%
1.5 - 1.99	30.0%	37.0 %	33.0%
2.0 - 2.99	36.0%	32.0 %	32.0%
3.0 - 3.99	56.0%	13.0 %	31.0%
4.0 - 5.99	82.0%	10.0 %	8.0%
6.0 - 15.0	100.0%	0.0 %	0.0%

only in parts to cut and paste operations. Finally, the column labeled as *Positive Clones* contains the percentage of functions clearly identified as cut and paste operations.

The matching process between two code fragments \mathcal{M} and \mathcal{P} is discussed with an example later in this section and is illustrated in Fig.3

The comparison cost function $C(i, j, \mathcal{M}, \mathcal{P})$ is the key factor in producing the final distance result when DP-based matching is used. There are many program features that can be considered to characterize a code fragment (indentation, keywords, metrics, uses and definitions of variables). Within the experimentation of this approach we used the following three different categories of features

1. definitions and uses of variables as well as, literal values within a statement:
 - (A) $Feature_1 : Statement \rightarrow String$ denotes the set of variables used in within a statement,
 - (B) $Feature_2 : Statement \rightarrow String$ denotes the set of variables defined within a statement
 - (C) $Feature_3 : Statement \rightarrow String$ denotes the set of literal values (i.e numbers, strings) within a statement (i.e. in a *printf* statement).
2. definitions and uses of data types :
 - (A) $Feature_1 : Statement \rightarrow String$ denotes the set of data type names used in within a statement,
 - (B) $Feature_2 : Statement \rightarrow String$ denotes the set of data type names defined within a statement

The comparison cost of the i th statement in the input \mathcal{P} and the j th statement of the model \mathcal{M} for the first two categories is calculated as :

$$\mathcal{C}(\mathcal{P}_i, \mathcal{M}_j) = \frac{1}{v} \cdot \sum_{m=1}^v \frac{\text{card}(\text{InputFeature}_m(\mathcal{P}_i) \cap \text{ModelFeature}_m(\mathcal{M}_j))}{\text{card}(\text{InputFeature}_m(\mathcal{P}_i) \cup \text{ModelFeature}_m(\mathcal{M}_j))} \quad (2)$$

where v is the size of the feature vector, or in other words how many features are used,

3. five metric values which are calculated compositionally from the statement level to function and file level :

The comparison cost of the i th statement in the input \mathcal{P} and the j th statement of the model \mathcal{M} when the five metrics are used is calculated as :

$$\mathcal{C}(\mathcal{P}_i, \mathcal{M}_j) = \sqrt{\sum_{k=1}^5 (M_k(\mathcal{P}_i) - M_k(\mathcal{M}_j))^2} \quad (3)$$

Within this framework new metrics and features can be used to make the comparison process more sensitive and accurate.

The following points on *insertion* and *deletion* costs need to be discussed.

- The *insertion* and *deletion* costs reflect the tolerance of the user towards partial matching (i.e. how much noise in terms of *insertions* and *deletions* is allowed before the matcher fails). Higher *insertion* and *deletion* costs indicate smaller tolerance, especially if cut-off thresholds are used (i.e. terminate matching if a certain threshold is exceeded), while smaller values indicate higher tolerance.
- The values for *insertion* and *deletion* should be higher than the threshold value by which two statements can be considered “*similar*”, otherwise an *insertion* or a *deletion* could be chosen instead of a *match*.
- A lower *insertion* cost than the corresponding *deletion* cost indicates the preference of the user to accept a code fragment \mathcal{P} that is written by inserting new statements to the model \mathcal{M} . The opposite holds when the *deletion* cost is lower than the corresponding *insertion* cost. A lower *deletion* cost indicates the preference of the user to accept a code fragment \mathcal{P} that is written by deleting statements from the model \mathcal{M} . *Insertion* and *deletion* costs are constant values throughout the comparison process and can be set empirically.

When different comparison criteria are used different distances are obtained. In Fig.2 (Clips) distances calculated using Dynamic Programming are shown for 138 pairs of functions (X - axis) that have been already identified as clones (i.e. zero distance) using the direct per function metric comparison. The dashed line shows distance results when definitions and uses of variables are used as features in the dynamic programming approach, while the solid line shows the distance results obtained when the five metrics are used as features. Note that in the Dynamic Programming based approach the metrics are used at

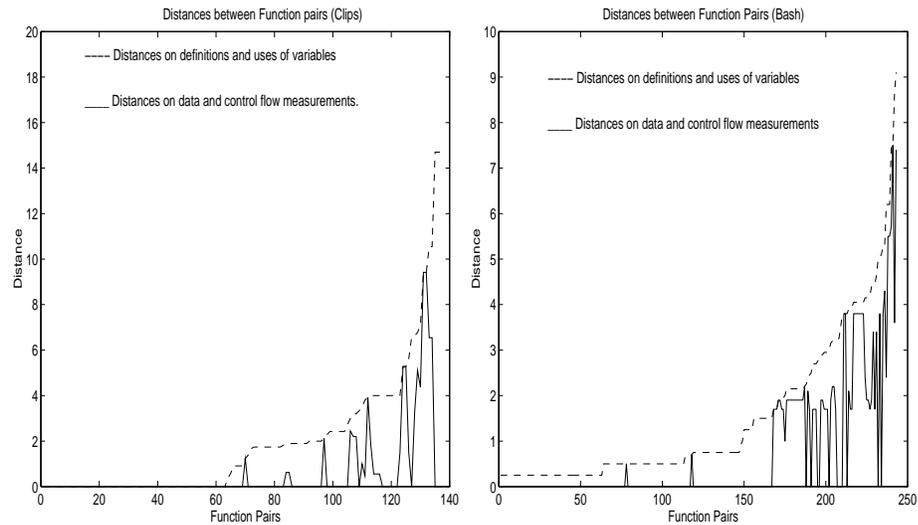


Figure 2. Distances between function pairs of possible function clones using DP-based matching.

the statement level, instead of the `begin-end` block level when metrics direct comparison is performed.

As an example consider the following statements \mathcal{M} and \mathcal{P} :

```
ptr = head;
while(ptr != NULL && !found)
{
    if(ptr->item == searchItem)
        found = 1
    else
        ptr = ptr->next;
}
```

```
while(ptr != NULL && !found)
{
    if(ptr->item == searchItem)
```

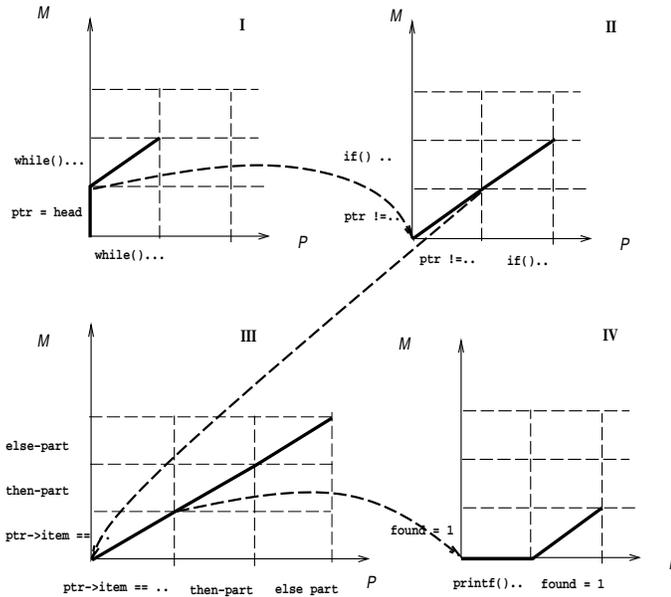


Figure 3. The matching process between two code fragments. Insertions are represented as horizontal lines and, deletions as vertical lines and, matches as diagonal lines.

```

{
    printf("ELEMENT FOUND : %s\n", searchItem);
    found = 1;
}
else
    ptr = ptr->next;
}

```

The Dynamic Programming matching based on definitions and uses of variables is illustrated in Fig. 3.

In the first grid the two code fragments are initially considered. At position (0, 0) of the first grid a deletion is considered as it gives the best cumulative distance to this point (assuming there will be a match at position (0, 1). The comparison of the two composite *while* statements in the first grid at position (0, 1), initiates a nested match (second grid). In the second grid the comparison of the composite *if-then-else* statements at position (1, 1) initiates a new nested match. In the third grid, the comparison of the composite *then-part* of the *if-then-else* statements initiates the final fourth nested match. Finally, in the fourth grid at position (0, 0), an insertion has been detected, as it gives the best cumulative distance to this point (assuming a potential match in (1, 0).

When a nested match process finishes it passes its result back to the position from which it was originally invoked and the matching continues from this point on.

3. Concept To Code Matching

The *concept assignment* (Biggerstaff, 1994) problem consists of assigning concepts described in a concept language to program fragments. Concept assignment can also be seen as a matching problem.

In our approach, concepts are represented as *abstract-descriptions* using a *concept language* called ACL. The intuitive idea is that a concept description may match with a number of different implementations. The probability that such a description matches with a code fragment is used to calculate a similarity measure between the description and the implementation. An *abstract-description* is parsed and a corresponding AST T_a is created. Similarly, source code is represented as an annotated AST T_c . Both T_a and T_c are transformed into a sequence of abstract and source code statements respectively using transformation rules. We use REFINE to build and transform both ASTs. The reason for this transformation is to reduce the complexity of the matching algorithm as T_a and T_c may have a very complex and different to each other structure. In this approach feature vectors of statements are matched instead of Abstract Syntax Trees. Moreover, the implementation of the Dynamic Programming algorithm is cleaner and faster once structural details of the ASTs have been abstracted and represented as sequences of entities.

The associated problems with matching concepts to code include :

- The choice of the conceptual language,
- The measure of similarity,
- The selection of a fragment in the code to be compared with the conceptual representation.

These problems are addressed in the following sections.

3.1. Language for Abstract Representation

A number of research teams have investigated and addressed the problem of code and plan localization. Current successful approaches include the use of graph grammars (Wills, 1992), (Rich, 1990), query pattern languages (Paul, 1994), (Muller, 1992), (Church, 1993), (Biggerstaff, 1994), sets of constraints between components to be retrieved (Ning, 1994), and summary relations between modules and data (Canfora, 1992).

In our approach a stochastic pattern matcher that allows for partial and approximate matching is used. A concept language specifies in an abstract way sequences of design concepts.

The concept language contains:

- Abstract statements \mathcal{S} that may match (generate) one or more statement types in the source code language. The correspondence between an *abstract statement* and the source code statements that it may generate is given at Table 2
- Abstract expressions \mathcal{E} that correspond to source code expression. The correspondence between an *abstract expression* and the source code expression that it may generate is given at Table 3
- Abstract feature descriptions \mathcal{F} that contain the feature vector data used for matching purposes. Currently the features that characterize an *abstract statement* and an *abstract expression* are:
 1. *Uses of variables* : variables that are used in a statement or expression
 2. *Definitions of variables*: ariables that are defined in a statement or expression
 3. *Keywords* : strings, numbers, characters that may used in the text of a code statement
 4. *Metrics* : a vector of five different complexity, data and control flow metrics.
- Typed Variables \mathcal{X}

Typed variables are used as a placeholders for feature vector values, when no actual values for the feature vector can be provided. An example is when we are looking for a *Traversal of a list* plan but we do not know the name of the pointer variable that exists in the code. A type variable can generate (match) with any actual variable in the source code provided that they belong to the same data type category. For example a *List* type abstract variable can be matched with an *Array* or a *Linked List node* source code pointer variable.

Currently the following abstract types are used :

 1. Numeral : Representing *Int*, and *float* types
 2. Character : Representing char types
 3. List : Representing *array* types
 4. Structure : Representing *struct* types
 5. Named : matching the actual data type name in the source code
- Operators \mathcal{O}

Operators are used to compose abstract statements in sequences. Currently the following operators have been defined in the language but only sequencing is implemented for the matching process :

 1. Sequencing (;) : To indicate one statement follows another
 2. Choice (\oplus) : To indicate choice (one or the other abstract statement will be used in the matching process)
 3. Inter Leaving (\parallel) : to indicate that two statements can be interleaved during the matching process

Table 2. Generation (Allowable Matching) of source code statements from ACL statements

<i>ACL Statement</i>	<i>Generated Code Statement</i>
<i>Abstract Iterative Statement</i>	<i>While Statement</i> <i>For Statement</i> <i>Do Statement</i>
<i>Abstract While Statement</i>	<i>While Statement</i>
<i>Abstract For Statement</i>	<i>For Statement</i>
<i>Abstract Do Statement</i>	<i>Do Statement</i>
<i>Abstract Conditional Statement</i>	<i>If Statement</i> <i>Switch Statement</i>
<i>Abstract If Statement</i>	<i>If Statement</i>
<i>Abstract Switch Statement</i>	<i>Switch Statement</i>
<i>Abstract Return Statement</i>	<i>Return Statement</i>
<i>Abstract GoTo Statement</i>	<i>GoTo Statement</i>
<i>Abstract Continue Statement</i>	<i>Continue Statement</i>
<i>Abstract Break Statement</i>	<i>Break Statement</i>
<i>Abstract Labeled Statement</i>	<i>Labeled Statement</i>
<i>AbstractStatement*</i>	Zero or more sequential source code statements
<i>AbstractStatement⁺</i>	One or more sequential source code statements

Table 3. Generation (Allowable Matching) of source code expressions from ACL expressions

<i>ACL Expression</i>	<i>Generated Code Expression</i>
<i>Abstract Function Call</i>	<i>Function Call</i>
<i>Abstract Equality</i>	<i>Equality (==)</i>
<i>Abstract Inequality</i>	<i>Inequality (! =)</i>
<i>Abstract Logical And</i>	<i>Logical And (&&)</i>
<i>Abstract Logical Or</i>	<i>Logical Or ()</i>
<i>Abstract Logical Not</i>	<i>Logical Not (!)</i>

- *Macros* \mathcal{M}

Macros are used to facilitate hierarchical plan recognition (Hartman, 1992), (Chikofsky, 19890). Macros are entities that refer to plans that are included at parse time. For example if a plan has been identified and is stored in the plan base, then special preprocessor statements can be used to include this plan to compose more complex patterns. Included plans are incorporated in the current pattern's AST at parse time. In this way they are similar to *inline* functions in C++.

Special macro definition statements in the Abstract Language are used to include the necessary macros.

Currently there are two types of macro related statements

1. **include definitions**: These are special statements in *ACL* that specify the name of the plan to be included and the file it is defined.

As an example consider the statement

```
include plan1.acl traversal-linked-list
```

that imports the plan *traversal-linked-list* defined in file *plan1.acl*.

2. **inline uses**: These are statements that direct the parser to *inline* the particular plan and include its AST in the original pattern's AST. As an example consider the inlining

```
plan: traversal-linked-list
```

that is used to include an instance of the *traversal-linked-list* plan at a particular point of the pattern. In a pattern more than one occurrence of an included plan may appear.

A typical example of a design concept in our concept language is given below. This pattern expresses an iterative statement (e.g. *while*, *for*, *do* loop that has in its condition an inequality expression that uses variable `?x` that is a pointer to the abstract type `list` (e.g. array, linked list) and the conditional expression contains the keyword "NULL". The body of `Iterative-Statement` contains a sequence of one or more statements (`+Statement`)

that uses at least variable `?y` (which matches to the variable `obj`) in the code below and contains the keyword `member`, and an `Assignment-Statement` that uses at least variable `?x`, defines variable `?x` which in this example matches to variable `field`, and contains the keyword `next`.

```
{
  Iterative-Statement(Inequality-Expression
                      abstract-description
                      uses : [ ?x : *list],
                      keywords : [ "NULL" ])
  {
    +-Statement
    abstract-description
    uses : [?y : string, ..]
    keywords : [ "member" ];
    Assignment-Statement
    abstract-description
    uses : [?x, ..],
    defines : [?x],
    keywords : [ "next" ]
  }
}
```

A code fragment that matches the pattern is:

```
{
while (field != NULL)
{
  if (!strcmp(obj,origObj) ||
      (!strcmp(field->AvalueType,"member") &&
       notInOrig ))
    if (strcmp(field->Avalue,"method") != 0)
      INSERT_THE_FACT(o->ATTLIST[num].Aname,origObj,
                    field->Avalue);
  field = field->nextValue;
}
}
```

3.2. *Concept-to-Code Distance Calculation*

In this section we discuss the mechanism that is used to match an abstract pattern given in *ACL* with source code.

In general the matching process contains the following steps :

1. Source code ($S_1; \dots S_k$) is parsed and an AST T_c is created.
2. The ACL pattern ($A_1; \dots A_n$) is parsed and an AST T_a is created.
3. A transformation program generates from T_a a Markov Model called Abstract Pattern Model (APM).
4. A Static Model called SCM provides the legal entities of the source language. The underlying finite-state automaton for the mapping between a *APM* state and an *SCM* state basically implements the Tables 2, 3.
5. Candidate source code sequences are selected.
6. A Viterbi (Viterbi, 1967) algorithm is used to find the best fit between the Dynamic Model and a code sequence selected from the candidate list.

A Markov model is a source of symbols characterized by states and transitions. A model can be in a state with certain probability. From a state, a transition to another state can be taken with a given probability. A transition is associated with the generation (recognition) of a symbol with a specific probability. The intuitive idea of using Markov models to drive the matching process is that an abstract pattern given in *ACL* may have many possible alternative ways to generate (match) a code fragment. A Markov model provides an appropriate mechanism to represent these alternative options and label the transitions with corresponding generation probabilities. Moreover, the Viterbi algorithm provides an efficient way to find the path that maximizes the overall generation (matching) probability among all the possible alternatives.

The selection of a code fragment to be matched with an abstract description is based on the following criteria : *a*) the first source code statement S_1 matches with the first pattern statement A_1 and, *b*) $S_2; S_3; \dots S_k$ belong to the innermost block containing S_1

The process starts by selecting all program blocks that match the criteria above. Once a candidate list of code fragments has been chosen the actual pattern matching takes place between the chosen statement and the outgoing transitions from the current active APM's state. If the type of the abstract statement the transition points to and the source code statement are compatible (compatibility is computed by examining the Static Model) then feature comparison takes place. This feature comparison is based on Dynamic Programming as described in section 2.3. A similarity measure is established by this comparison between the features of the abstract statement and the features of the source code statement. If composite statements are to be compared, an expansion function "flattens" the structure by decomposing the statement into a sequence of its components. For example an `if` statement will be decomposed as a sequence of an `expression` (for its condition), its `then` part and its `else` part. Composite statements generate nested matching sessions as in the DP-based code-to-code matching.

3.3. ACL Markov Model Generation

Let T_c be the AST of the code fragment and T_a be the AST of the abstract representation.

A measure of similarity between T_c and T_a is the following probability

$$P_r(T_c|T_a) = P_r(r_{c_1}, \dots, r_{c_i}, \dots, r_{c_l} | r_{a_1}, \dots, r_{a_i}, \dots, r_{a_L}) \quad (4)$$

where,

$$(r_{c_1}, \dots, r_{c_i}, \dots, r_{c_l}) \quad (5)$$

is the sequence of the grammar rules used for generating T_c and

$$(r_{a_1}, \dots, r_{a_i}, \dots, r_{a_L}) \quad (6)$$

is the sequence of rules used for generating T_a . The probability in (1) cannot be computed in practice, because of complexity issues related to possible variations in T_a generating T_c . An approximation of (4) is thus introduced.

Let S_1, \dots, S_k be a sequence of program statements During the parsing that generates T_a , a sequence of abstract descriptions is produced. Each of these descriptions is considered as a Markov source whose transitions are labeled by symbols A_j which in turn generate (match) source code.

The sequence of abstract descriptions A_j forms a pattern \mathcal{A} in Abstract Code Language (ACL) and is used to build dynamically a Markov model called Abstract Pattern Model (APM). An example of which is given in Fig.4.

The Abstract Pattern Model is generated an ACL pattern is parsed. Nodes in the APM represent Abstract ACL Statements and arcs represent transitions that determine what is expected to be matched from the source code via a link to a static, permanently available Markov model called a Source Code Model (SCM).

The Source Code Model is an alternative way to represent the syntax of a language entity and the correspondence of Abstract Statements in ACL with source code statements.

For example a transition in APM labeled as (pointing to) an Abstract While Statement is linked with the while node of the static model. In its turn a while node in the SCM describes in terms of states and transitions the syntax of a legal while statement in C.

The best alignment between a sequence of statements $\mathcal{S} = S_1; S_2; S_k$ and a pattern $\mathcal{A} = A_1; A_2; \dots; A_j$ is computed by the Viterbi (Viterbi, 1967) dynamic programming algorithm using the SCM and a feature vector comparison function for evaluating the following type of probabilities:

$$P_r(S_1, S_2, \dots, S_i | A_{f(i)}) \quad (7)$$

where $f(i)$ indicates which abstract description is allowed to be considered at step i . This is determined by examining the reachable APM transitions at the i th step. For the matching to succeed the constraint $P_r(S_1 | A_1) = 1.0$ must be satisfied and $A_{f(k)}$ corresponds to a final APM state.

This corresponds to approximating (4) as follows (Brown, 1992):

$$P_r(T_c|T_a) \simeq P_r(S_1; \dots; S_k | A_1; \dots; A_n) =$$

$$\prod_{i=1}^k \max(P_r(S_1; S_2 \dots S_{i-1} | A_1; A_2; \dots A_{f(i-1)}) \bullet P_r(S_i | A_{f(i)})) \quad (8)$$

This is similar to the code-to-code matching. The difference is that instead of matching source code features, we allow matching abstract description features with source code features. The dynamic model (APM) guarantees that only the allowable sequences of comparisons are considered at every step.

The way to calculate similarities between individual abstract statements and code fragments is given in terms of probabilities of the form $P_r(S_i | A_j)$ as the probability of abstract statement A_j generating statement S_i .

The probability $p = P_r(S_i | A_j) = P_{scm}(S_i | A_j) * P_{comp}(S_i | A_j)$ is interpreted as “The probability that code statement S_i can be generated by abstract statement A_j ”. The magnitude of the logarithm of the probability p is then taken to be the distance between S_i and A_j .

The value of p is computed by multiplying the probability associated with the corresponding state for A_j in SCM with the result of comparing the feature vectors of S_i and A_j . The feature vector comparison function is discussed in the following subsection.

As an example consider the APM of Fig. 4 generated by the pattern $A_1; A_2^*; A_3^*$, where A_j is one of the legal statements in ACL. Then the following probabilities are computed for a selected candidate code fragment S_1, S_2, S_3 :

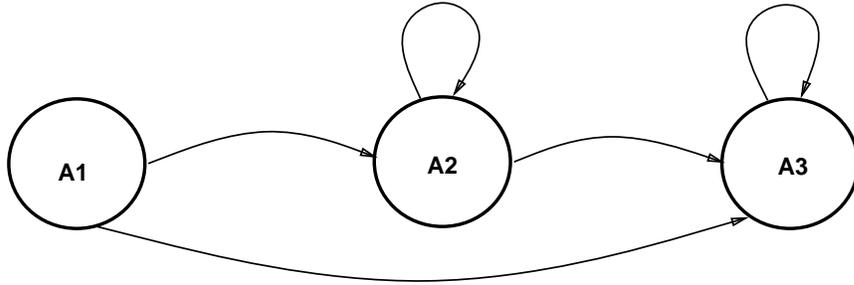


Figure 4. A dynamic model for the pattern $A_1; A_2^*; A_3^*$

$$P_r(S_1 | A_1) = 1.0 \quad (\text{delineation criterion}) \quad (9)$$

$$P_r(S_1, S_2 | A_2) = P_r(S_1 | A_1) \cdot P_r(S_2 | A_2) \quad (10)$$

$$P_r(S_1, S_2 | A_3) = P_r(S_1 | A_1) \cdot P_r(S_2 | A_3) \quad (11)$$

$$P_r(S_1, S_2, S_3 | A_3) = \text{Max} \begin{cases} P_r(S_1, S_2 | A_2) \cdot P_r(S_3 | A_3) \\ P_r(S_1, S_2 | A_3) \cdot P_r(S_3 | A_3) \end{cases} \quad (12)$$

$$P_r(S_1, S_2, S_3|A_2) = P_r(S_1, S_2|A_2) \cdot P_r(S_3|A_2) \quad (13)$$

Note that when the first two program statements S_1, S_2 have already been matched, (equations 12 and 13) two transitions have been consumed and the reachable active states currently are A2 or A3.

Moreover at every step the probabilities of the previous steps are stored and there is no need to be reevaluated. For example $P_r(S_1, S_2|A_2)$ is computed in terms of $P_r(S_1|A_1)$ which is available from the previous step.

With each transition we can associate a list of probabilities based on the type of expression likely to be found in the code for the plan that we consider.

For example, in the `Traversal` of a linked list plan the `while` loop condition, which is an expression, most probably generates an *inequality* of the form (*list-node-ptr* \neq *NULL*) which contains an identifier reference and the keyword `NULL`.

An example of a static model for the `pattern-expression` is given in Fig. 5. Here we assume for simplicity that only four C expressions can be generated by a `Pattern-Expression`.

The initial probabilities in the static model are provided by the user who either may give a uniform distribution in all outgoing transitions from a given state or provide some subjectively estimated values. These values may come from the knowledge that a given plan is implemented in a specific way. In the above mentioned example of the `Traversal` of a linked list plan the `Iterative-Statement` pattern usually is implemented with a `while` loop. In such a scenario the `Iterative` abstract statement can be considered to generate a `while` statement with higher probability than a `for` statement. Similarly, the expression in the *while* loop is more likely to be an *inequality* (Fig. 5). The preferred probabilities can be specified by the user while he or she is formulating the query using the ACL primitives. Once the system is used and results are evaluated these probabilities can be adjusted to improve the performance.

Probabilities can be dynamically adapted to a specific software system using a cache memory method originally proposed (for a different application) in (Kuhn, 1990).

A cache is used to maintain the counts for most frequently recurring statement patterns in the code being examined. Static probabilities can be weighted with dynamically estimated ones as follows :

$$P_{scm}(S_i|A_j) = \lambda \cdot P_{cache}(S_i|A_j) + (1 - \lambda) \cdot P_{static}(S_i|A_j) \quad (14)$$

In this formula $P_{cache}(S_i|A_j)$ represents the frequency that A_j generates S_i in the code examined at run time while $P_{static}(S_i|A_j)$ represents the a-priori probability of A_j generating S_i given in the static model. λ is a weighting factor. The choice of the weighting factor λ indicates user's preference on what weight he or she wants to give to the feature vector comparison. Higher λ values indicate a stronger preference to depend on feature vector comparison. Lower λ values indicate preference to match on the type of statement and not on the feature vector.

The value of λ can be computed by deleted-interpolation as suggested in (Kuhn, 1990). It can also be empirically set to be proportional to the amount of data stored in the cache.

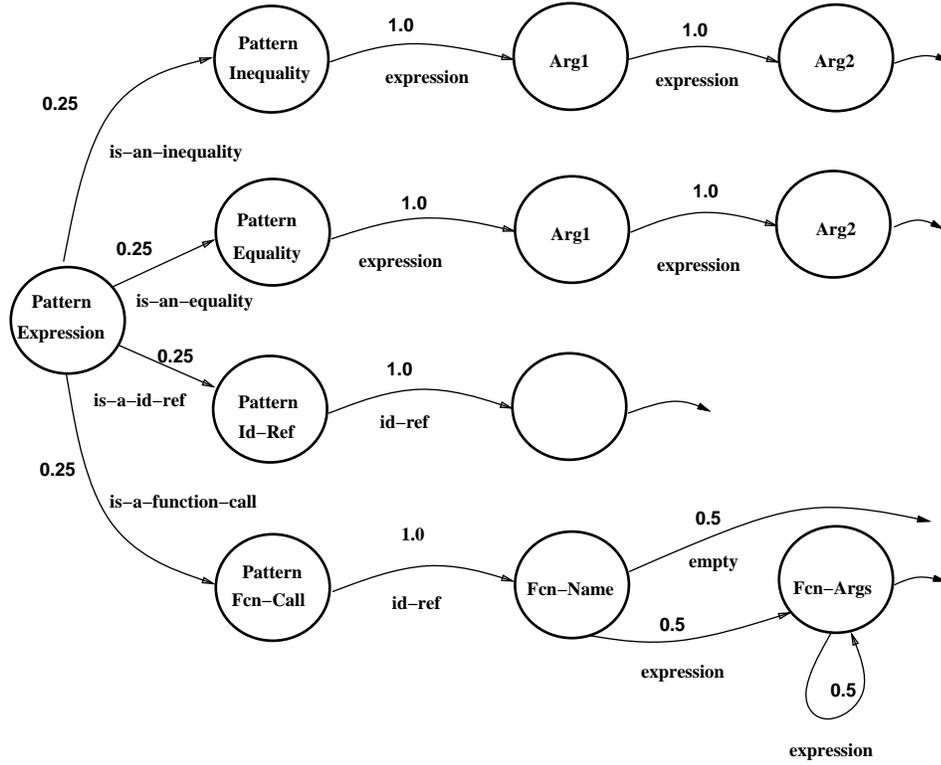


Figure 5. The static model for the expression-pattern. Different transition probability values may be set by the user for different plans. For example the *traversal of linked-list* plan may have higher probability attached to the *is-an-inequality* transition as the programmer expects a pattern of the form $(field \neq NULL)$

As proposed in (Kuhn, 1990), different cache memories can be introduced, one for each A_j . Specific values of λ can also be used for each cache.

3.4. Feature Vector Comparison

In this section we discuss the mechanism used for calculating the similarity between two feature vectors. Note that S_i 's and A_j 's feature vectors are represented as annotations in the corresponding ASTs.

The feature vector comparison of S_i, A_j returns a value $p = P_r(S_i|A_j)$.

The features used for comparing two entities (source and abstract) are:

1. *Variables defined* $\mathcal{D} : \text{Source-Entity} \rightarrow \{\text{String}\}$
2. *Variables used* $\mathcal{U} : \text{Source-Entity} \rightarrow \{\text{String}\}$

3. *Keywords* $\mathcal{K} : \text{Source-Entity} \rightarrow \{\text{String}\}$

4. *Metrics*

- Fan out $\mathcal{M}_1 : \text{Source-Entity} \rightarrow \text{Number}$
- D-Complexity $\mathcal{M}_2 : \text{Source-Entity} \rightarrow \text{Number}$
- McCabe $\mathcal{M}_3 : \text{Source-Entity} \rightarrow \text{Number}$
- Albrecht $\mathcal{M}_4 : \text{Source-Entity} \rightarrow \text{Number}$
- Kafura $\mathcal{M}_5 : \text{Source-Entity} \rightarrow \text{Number}$

These features are AST annotations and are implemented as mappings from an AST node to a set of AST nodes, set of Strings or set of Numbers.

Let S_i be a source code *statement* or *expression* in program \mathcal{C} and A_j an *abstract statement* or *expression* in pattern \mathcal{A} . Let the feature vector associated with S_i be \mathcal{V}_i and the feature vector associated with A_j be \mathcal{V}_j . Within this framework we experimented with the following similarity considered in the computation as a probability:

$$P_{comp}(S_i|A_j) = \frac{1}{v} \cdot \sum_{n=1}^v \frac{\text{card}(\text{AbstractFeature}_{j,n} \cap \text{CodeFeature}_{i,n})}{\text{card}(\text{AbstractFeature}_{j,n} \cup \text{CodeFeature}_{i,n})} \quad (15)$$

where v is the size of the feature vector, or in other words how many features are used, $\text{CodeFeature}_{i,n}$ is the n th feature of source statement S_i and, $\text{AbstractFeature}_{j,n}$ is the n th feature of the ACL statement A_j .

As in the code to code dynamic programming matching, lexicographical distances between variable names (i.e. next, next value) and numerical distances between metrics are used when no exact matching is the objective. Within this context two strings are considered similar if their lexicographical distance is less than a selected threshold, and the comparison of an abstract entity with a code entity is valid if their corresponding metric values are less than a given threshold.

These themes show that ACL is viewed more as a vehicle where new features and new requirements can be added and be considered for the matching process. For example a new feature may be a link or invocation to another pattern matcher (i.e. SCRUPLE) so that the abstract pattern in ACL succeeds to match a source code entity if the additional pattern matcher succeeds *and* the rest of the feature vectors match.

4. System Architecture

The concept-to-code pattern matcher of the Ariadne system is composed of four modules.

The first module consists of an abstract code language (ACL) and its corresponding parser. Such a parser builds at run time, an AST for the ACL pattern provided by the user. The ACL AST is built using Refine and its corresponding domain model maps to entities of the C language domain model. For example, an *Abstract-Iterative-Statement* corresponds to an *Iterative-Statement* in the C domain model.

A static explicit mapping between the ACL's domain model and C's domain model is given by the SCM (Source Code Model), Ariadne's second module. SCM consists of states and transitions. States represent Abstract Statements and are nodes of the ACL's AST. Incoming transitions represent the nodes of the C language AST that can be matched by this Abstract Statement. Transitions have initially attached probability values which follow a uniform distribution. A subpart of the SCM is illustrated in Fig. 5 where it is assumed for simplicity that an Abstract Pattern Expression can be matched by a C inequality, equality, identifier reference, and a function call.

The third module builds the Abstract Pattern Model at run time for every pattern provided by the user. APM consists of states and transitions. States represent nodes of the ACL's AST. Transitions model the structure of the pattern given, and provide the pattern statements to be considered for the next matching step. This model directly reflects the structure of the pattern provided by the user. Formally APM is an automaton $\langle Q, \Sigma, \delta, q_0, F \rangle$ where

- Q , is the set of states, taken from the domain of ACL's AST nodes
- Σ , is the input alphabet which consists of nodes of the C language AST
- δ , is a transition function implementing statement expansion (in the case of composite abstract or C statements) and the matching process
- q_0 , is the Initial state. The set of outgoing transitions must match the first statement in the code segment considered.
- F , is a set of final states. The matching process stops when one of the final states have been reached and no more statements from the source code can be matched.

Finally, the fourth module is the matching engine. The algorithm starts by selecting candidate code fragments $\mathcal{P} = S_1; S_2; \dots S_k$, given a model $\mathcal{M} = A_1; A_2; \dots A_n$.

The Viterbi algorithm is used to evaluate the best path from the start to the final state of the APM.

An example of a match between two simple expressions (a *function call* and an *Abstract-Expression* is given below :

```
INSERT_THE_FACT(o->ATTLIST[num].Aname,origObj,
                field->Avalue);
```

is matched with the abstract pattern

```
Expression(abstract-description
           uses : [ "ATTLIST", "Aname", "Avalue" ]
           Keywords : [ "INSERT", "FACT" ] )
```

In this scenario both abstract and code statements are simple and do not need expansion. `Expression` and `INSERT_THE_FACT(...)` are type compatible statements because an expression can generate a function call (Fig. 5) so the matching can proceed. The next step is to compare features, and lexicographical distances between variable names in the abstract and source statement. The final value is obtained by multiplying the value obtained from the feature vectors comparison and the probability that *Expression* generates a *Function Call*.

As the pattern statement does not specify what type of expression is to be matched the static model (SCM) provides an estimate. In the SCM given in Fig. 5 the likelihood that the `Expression` generates a function call is 0.25. The user may provide such a value if a plan favours a particular type instead of another. For example in the `Traversal of a linked list` plan the loop statement is most likely to be a `while` loop. Once a final value is set then a record $\langle \text{abstract_pattern}, \text{matched_code}, \text{distance_value} \rangle$ is created and is associated with the relevant transition of the APM. The process ends when a final state of the APM has been reached and no more statements match the pattern.

With this approach the matching process does not fail when imperfect matching between the pattern and the code occurs. Instead, partial and inexact matching can be computed. This is very important as the programmer may not know how to specify in detail the code fragment that is sought.

To reduce complexity when variables in the pattern statement occur, Ariadne maintains a global binding table and it checks if the given pattern variable is bound to one of the legal values from previous instantiations. These legal values are provided by the binding table and are initialized every time a new pattern is tried and a new APM is created.

5. Conclusion

Pattern matching plays an important role for plan recognition and design recovery. In this paper we have presented a number of pattern matching techniques that are used for code-to-code and concept-to-code matching. The main objective of this research was to devise methods and algorithms that are time efficient, allow for partial and inexact matching, and tolerate a measure of dissimilarity between two code fragments. For code representation schemes the program's Abstract Syntax Tree was used because it maintains all necessary information without creating subjective views of the source code (control or data flow biased views).

Code-to-code matching is used for clone detection and for computing similarity distances between two code fragments. It is based on *a*) a dynamic programming pattern matcher that computes the best alignment between two code fragments and *b*) metric values obtained for every expression, statement, and block of the AST. Metrics are calculated by taking into account a number of control and data program properties. The dynamic programming pattern matcher produces more accurate results but the metrics approach is cheaper and can be used to limit the search space when code fragments are selected for comparison using the dynamic programming approach.

We have experimented with different code features for comparing code statements and are able to detect clones in large software systems ≥ 300 KLOC. Moreover, clone detection is used to identify "conceptually" related operations in the source code. The performance

is limited by the fact we are using a LISP environment (frequent garbage collection calls) and the fact that metrics have to be calculated first. When the algorithm using metric values for comparing program code fragments was rewritten in C it performed very well. For 30KLOCS of the CLIPS system and for selecting candidate clones from approximately 500,000 pairs of functions the C version of the clone detection system run in less than 10 seconds on a Sparc 10, as opposed to a Lisp implementation that took 1.5 minutes to complete. The corresponding DP-based algorithm implemented in Lisp took 3.9 minutes to complete.

Currently the system is used for system clustering, redocumentation and program understanding. Clone detection analysis reveals clusters of functions with similar behaviour suggesting thus a possible system decomposition. This analysis is combined with other data flow analysis tools (Konto, 1994) to obtain a multiple system decomposition view. For the visualization and clustering aspect the Rigi tool developed at the University of Victoria is used. Integration between the Ariadne tool and the Rigi tool is achieved via the global software repository developed at the University of Toronto.

The false alarms using only the metric comparison was on average for the three systems 39% of the total matches reported. When the DP approach was used, this ratio dropped to approximately 10% in average (when zero distance is reported). Even if the noise presents a significant percentage of the result, it can be filtered in almost all cases by adding new metrics (i.e. line numbers, Halstead's metric, statement count). The significant gain though in this approach is that we can limit the search space to a few hundreds (or less than a hundred, when DP is considered) of code fragment pairs from a pool of half a million possible pairs that could have been considered in total. Moreover, the method is fully automatic, does not require any knowledge of the system and is acceptable computationally $\mathcal{O}(n * m)$ for DP, where m is the size of the model and n the size of the input.

Concept-to-code matching uses an abstract language (ACL) to represent code operations at an abstract level. Markov models and the Viterbi algorithm are used to compute similarity measures between an abstract statement and a code statement in terms of the probability that an abstract statement generates the particular code statement.

The ACL can be viewed not only as a regular expression-like language but also as a vehicle to gather query features and an engine to perform matching between two artifacts. New features, or invocations and results from other pattern matching tools, can be added to the features of the language as requirements for the matching process. A problem we foresee arises when binding variables exist in the pattern. If the pattern is vague then complexity issues slow down the matching process. The way we have currently overcome this problem is for every new binding to check only if it is a legal one in a set of possible ones instead of forcing different alternatives when the matching occurs.

Our current research efforts are focusing on the development of a generic pattern matcher which given a set of features, an abstract pattern language, and an input code fragment can provide a similarity measure between an abstract pattern and the input stream.

Such a pattern matcher can be used *a)* for retrieving plans and other algorithmic structures from a variety of large software systems (aiding software maintenance and program understanding), *b)* querying digital databases that may contain partial descriptions of data and *c)* recognizing concepts and other formalisms in plain or structured text (e.g., HTML)

Another area of research is the use of metrics for finding a measure of the changes introduced from one to another version in an evolving software system. Moreover, we investigate the use of the cloning detection technique to identify similar operations on specific data types so that generic classes and corresponding member functions can be created when migrating a procedural system to an object oriented system.

Notes

1. In this paper, "reverse engineering" and related terms refer to legitimate maintenance activities based on source-language programs. The terms do not refer to illegal or unethical activities such as the reverse compilation of object code to produce a competing product.
2. "The Software Refinery" and REFINE are trademarks of Reasoning Systems, Inc.
3. We are using a commercial tool called REFINE (a trademark of Reasoning Systems Corp.).
4. The Spearman-Pearson rank correlation test was used.

References

- Adamov, R. "Literature review on software metrics", Zurich: Institut fur Informatik der Universitat Zurich, 1987.
- 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 1995
- Biggerstaff, T., Mitbander, B., Webster, D., "Program Understanding and the Concept Assignment Problem", *Communications of the ACM*, May 1994, Vol. 37, No.5, pp. 73-83.
- P. Brown et. al. "Class-Based n-gram Models of natural Language", *Journal of Computational Linguistics*, Vol. 18, No.4, December 1992, pp.467-479.
- Buss, E., et. al. "Investigating Reverse Engineering Technologies for the CAS Program Understanding Project", *IBM Systems Journal*, Vol. 33, No. 3, 1994, pp. 477-500.
- G. Canfora., A. Cimitile., U. Carlini., "A Logic-Based Approach to Reverse Engineering Tools Production" *Transactions of Software Engineering*, Vol.18, No. 12, December 1992, pp. 1053-1063.
- Chikofsky, E.J. and Cross, J.H. II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, Jan. 1990, pp. 13 - 17.
- Church, K., Helfman, I., "Dotplot: a program for exploring self-similarity in millions of lines of text and code", *J. Computational and Graphical Statistics 2,2*, June 1993, pp. 153-174.
- C-Language Integrated Production System *User's Manual* NASA Software Technology Division, Johnson Space Center, Houston, TX.
- Fenton, E. "Software metrics: a rigorous approach", Chapman and Hall, 1991.
- Halstead, M., H., "Elements of Software Science", New York: Elsevier North-Holland, 1977.
- J. Hartman., "Technical Introduction to the First Workshop on Artificial Intelligence and Automated Program Understanding" *First Workshop on AI and Automated Program Understanding*, AAAI'92, San-Jose, CA.
- Horwitz S., "Identifying the semantic and textual differences between two versions of a program. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1990, pp. 234-245.
- Jankowitz, H., T., "Detecting plagiarism in student PASCAL programs", *Computer Journal*, 31.1, 1988, pp. 1-8.
- Johnson, H., "Identifying Redundancy in Source Code Using Fingerprints" In *Proceedings of CASCON '93*, IBM Centre for Advanced Studies, October 24 - 28, Toronto, Vol.1, pp. 171 - 183.
- Kuhn, R., DeMori, R., "A Cache-Based Natural Language Model for Speech Recognition", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 12, No.6, June 1990, pp. 570-583.
- Kontogiannis, K., DeMori, R., Bernstein, M., Merlo, E., "Localization of Design Concepts in Legacy Systems", In *Proceedings of International Conference on Software Maintenance 1994*, September 1994, Victoria, BC. Canada, pp. 414-423.

- Kontogiannis, K., DeMori, R., Bernstein, M., Galler, M., Merlo, E., "Pattern matching for Design Concept Localization", *In Proceedings of the Second Working Conference on Reverse Engineering*, July 1995, Toronto, ON, Canada, pp. 96-103.
- McCabe T., J. "Reverse Engineering, reusability, redundancy : the connection", *American Programmer* 3, 10, October 1990, pp. 8-13.
- Moller, K., Software metrics: a practitioner's guide to improved product development"
- Muller, H., Corrie, B., Tilley, S., *Spatial and Visual Representations of Software Structures*, Tech. Rep. TR-74. 086, IBM Canada Ltd. April 1992.
- Mylopoulos, J., "Telos : A Language for Representing Knowledge About Information Systems," *University of Toronto, Dept. of Computer Science Technical Report KRR-TR-89-1*, August 1990, Toronto.
- J. Ning., A. Engberts., W. Kozaczynski., "Automated Support for Legacy Code Understanding", *Communications of the ACM*, May 1994, Vol.37, No.5, pp.50-57.
- 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.
- Rich, C. and Wills, L.M., "Recognizing a Program's Design: A Graph-Parsing Approach," *IEEE Software*, Jan 1990, pp. 82 - 89.
- Tilley, S., Muller, H., Whitney, M., Wong, K., "Domain-retargetable Reverse EngineeringII: Personalized User Interfaces", *In CSM'94 : Proceedings of the 1994 Conference on Software Maintenance*, September 1994, pp. 336 - 342.
- Viterbi, A.J., "Error Bounds for Convolutional Codes and an Asymptotic Optimum Decoding Algorithm", *IEEE Trans. Information Theory*, 13(2) 1967.
- Wills, L.M., "Automated Program Recognition by Graph Parsing", *MIT Technical Report, AI Lab No. 1358*, 1992