Pattern Matching for Design Concept Localization

K. Kontogiannis† R. DeMori† M. Bernstein†
M. Galler† Ettore Merlo*

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

*Ecole Polytechnique de Montréal
Dept. de Genie Electrique Genie Informatique,
6079, Succ. Centre Ville, Montréal, Quebec, Canada

Abstract

The effective synergy of a number of different techniques is the key to the successful development of an efficient Reverse Engineering environment. Compiler technology, pattern matching techniques, visualization tools, and software repositories play an important role for the identification of procedural, data, and abstract-data-type related concepts in the source code. This paper describes a number of techniques used for the development of a distributed reverse engineering environment. Design recovery is investigated through code-to-code and abstract-descriptions-to-code pattern matching techniques used to locate code that may implement a particular plan or algorithm. The code-to-code matching uses dynamic programming techniques to locate similar code fragments and is targeted for large software systems (1MLOC). Patterns are specified either as source code or as a sequence of abstract statements written in an concept language developed for this purpose. Markov models are used to compute similarity measures between an abstract description and a code fragment in terms of the probability that a given abstract statement can generate a given code fragment. The abstract-description-to-code matcher is under implementation and early experiments show it is a promising technique.

*This work is funded by IBM Canada Ltd. Laboratory - Center for Advanced Studies (Toronto), National Research Council of Canada, and McGill University.

1 Introduction

Program Understanding is the process of examination of a software system in order to develop representations of the system's intended architecture, behaviour, and purpose. The identification and recognition of concepts related to sequences of computations, permanent relationships between program's data, and operations on classes of data types are focal points in a reverse engineering environment.

Recognition of sequences of computations is examined through code-to-code and concept-to-code matching. Permanent relationships between data, is investigated through clustering based on data flow properties. Finaly, code cloning detection may reveal important operations on data types, identify system clusters, help reorganize the system in an object-oriented way, and suggest points where potential errors can be found.

For this purpose, an approach is proposed in which, source code is parsed and represented as an annotated Abstract Syntax Tree (AST). Nodes in the AST represent language components (e.g statements, expressions) and arcs represent relationships between these language components. For example an `IF`-statement is represented as an AST node with three arcs pointing to the `condition`, the `THEN`-part, and the `ELSE`-part.

During the analysis phase each node is annotated with control and data flow information. It can be a vector of software metrics [3], a set of data bindings with the rest of the system, [9], or a set of keywords, variable names and data types. The REFINE [1] environment is used to analyze and store the AST and

[1]REFINE is a trademark of Reasoning Systems

0-8186-7111-4/95 $4.00 © 1995 IEEE

its annotations. The annotations are computed in a compositional way from the leaves to the root of the AST.

Two problems are considered in the following sections. The first is based on localizing similar code fragments and we refer to it as *code-to-code* matching. It is used for example to perform code duplication detection in large C software systems.

Code-to-code matching is achieved by using a dynamic programming (DP) based pattern matcher that determines the best alignment between two code fragments at the statement level. 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 such an alignment.

The second problem, abstract-description-to-code matching, has a more ambitious objective. In this case, pattern matcher is driven by a Markov model that describes the similarity between an abstract pattern and a code fragment as a set of probabilities that a given abstract statement can generate a given source statement.

2 Code-To-Code Matching

While the general problem of determining if two functions are the same (i.e. produce the same outputs given the same inputs) is known to be undecidable, clone detection is a computable approximation in which a measure of structural similarity is used to determine if two fragments are in some way equivalent. What makes clone detection difficult is that replicas of the same original code, will be somewhat modified and it is necessary to definse a metric for computing a "degree of similarity". Possible modifications which are captured by the feature vectors we consider and the matching process include *a)* deletion of statements, *b)* insertion of new statements and, *c)* modification of existing statements.

The process starts by transforming the AST that corresponds to the two code fragments into two sequences of expressions and statements. For example, an `if`-statement will be expanded into the sequence [`condition`-expression, `THEN`-part, `ELSE`-part].

The comparison at the statement level is based on a set of features that are calculated during the AST annotation phase. Program features [2] used in the matching process include *(a)* software quality and complexity metrics [1] (Fanout, D-Complexity, McCabe Cyclomatic Complexity, a variation of Albrecht's U.F.C metric, and a variation of Henry-Kafura's I.F metric) *(b)* Global/local variables set/used *(c)* Files accessed (e.g. *fscanf, fprintf* statements) *(d)* Type and number of formal parameters passed by reference *(e)* Data bindings, and *(f)* Keywords and associated concepts

All the features can be calculated in a compositional way starting from the leaves and working upwards to the root of the AST. With this technique we are able to obtain and classify program features for every node in the AST. In this way we can examine and compare at any level of granularity (statement, block, function, module) data and control flow properties either as metric values or as AST annotations.

One may ask why these program features have been selected and what other alternative program features could have been used instead. The selection of these program features is based on the type of internal software attributes that are *orthogonal* and *fundamental*, as well as being widely and commonly accepted as attributes that classify software quality, structure, development effort, and maintainability. These attributes include:

- Program length (number of comment lines, and non-comment lines)
- Functionality (the amount of "function" a code fragment delivers
- Complexity (a measure of how difficult a program is to test and maintain)
- Redundancy (a measure of code cloning, and dead code)
- Reuse (an indicator of how many times a particular function or code fragment is used in the system, either via a function call or as a cut-and-paste operation)

Additional features that may have been used are annotations based on alias information, value range analysis, redundant expression elimination, data dependency graphs, and slicing. The reason we have not included these features in the initial analysis (*Reverse Engineering in the Large*) is that they are very expensive to be calculated for large (10KLOC) code fragments. Instead we want to provide tools that focus attention on smaller program components so that the maintainer can apply well established techniques (static data flow analysis, debuggers, profilers) to understand their functionality – *Reverse Engineering in the Small.*

[2] Other features can be used too.

As it is not always possible to have a complete plan base, the user can select at run time a sequence of statements to be used as a *model* (e.g. P_1, P_2, P_3, P_4, P_5 in Fig.1). The matching process will find similar code fragments that may implement the same algorithm or plan as the model selected.

For calculating the distance between two code fragments we have investigated the idea of using a dynamic programming based algorithm that computes the best alignment between two code fragments in terms of *insertions*, *deletions*, and *substitutions* of individual expressions and statements in the two code fragments compared. 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 : *Feature-Vector X Feature-Vector* $\rightarrow$ *Real*

where:

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

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 between positions i and j in code fragment Q
- $D(x,y,\mathcal{P},\mathcal{M})$ is the the distance between the features **x** of the code fragment P and features **y** of the the model M ,
- $\Delta(i,j,\mathcal{P},\mathcal{M})$ is the cost of deleting the j*th* statement of M, at the i*th* statement of the fragment P
- $I(i,j,\mathcal{P},\mathcal{M})$ the cost of inserting the i*th* statement of P at the j*th* statement of the model M and
- $C(i,j,\mathcal{P},\mathcal{M})$ is the comparison cost that of the i*th* statement of the code fragment P and the j*th* statement of the model 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.

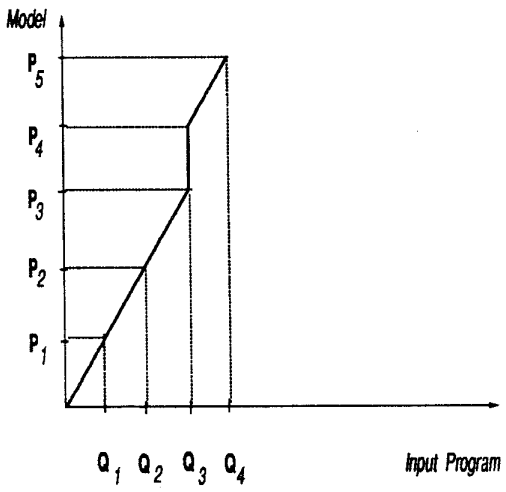

Figure 1: The matching process between two code fragments. The diagonal in position P_1, Q_1 represents a match between statement P_1, Q_1, while the vertical at position P_4, Q_3 the deletion of statement Q_3in of the input fragment at position P_4 of the model.

Note that *insertion* and *deletion* costs are used by the dynamic programming algorithm to calculate the best fit between two code fragments. An intuitive interpreataion of the best fit using *insertions* and *deletions* is "if we insert (delete) statement i (j) of the input (model) at position j (i) of the model (input) then the model and the input have the smallest 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.

Fig1. illustrates the warping function of the matching process between fragments P consisting of the individual statements P_1, P_2, P_3, P_4, P_5 (used as the model) and Q consisting of the individual statements Q_1, Q_2, Q_3, Q_4, representing a source code fragment to be matched against the model P.

As an example consider clone detection in the source code of the expert system shell CLIPS [5] that consists of approximately 40KLOC distributed in 41 files and 700 functions. Code cloning and similarity detection at the function level revealed afrer investigation 70 related clusters of potential clones. In 60 clusters elements found to implement related concepts within each cluster. Examples of concepts found include *print routines for object hierarchies*, *assert-*

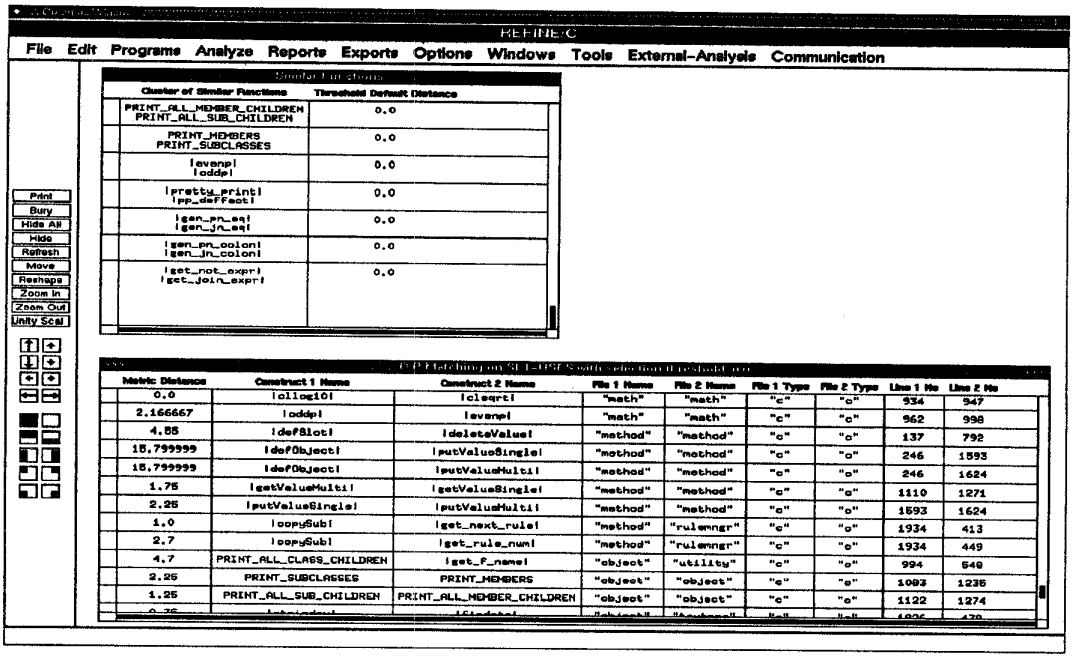

Figure 2: Dynamic Programming based code-to-code matching. Code cloning detection reveals collections of functions that correspond to same class of "functionality" or procedural concepts. In the upper part collections of functions related to *print* concepts are shown. In the lower part pairs of functions related to *accessing/updating* attribute values is shown.

ing/deleting elements in the fact list, debugging commands, rule and fact management functions (e.g. acivate and maintain the rule agenda, add and delete facts), *retrieve/update/delete attribute values* etc. Our experiments have shown that code cloning detection can be used for initial system clustering as a useful redocumentation technique that does not require *apriori* knowledge of the system. In Fig.2 collections of functions that have been identified as clones reveal associated concepts relating with *Print routines for object hierarchies* and *pretty print* routines (upper part), or *accessing attribute values* routines (lower part). Moreover, using different feature vectors for comparison we obtain different results. Function pairs with disimilarity measure 0.0 (upper part) may have different disimilarity measures (lower part) when additional features are presented (e.g lexicographical distances in variable names).

3 Concept-to-Code Matching

The *concept assignment* [2] 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*. 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 associated problems with matching concepts with 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.

4 Language for Abstract Representation

A number of research teams have identified the problem of code localization using query pattern languages, pattern matching on program text, and pattern matching on control and data flow graphs [13], [11], [4], [6], [14]. In our approach we give weight to a stochastic pattern matcher that allows for partial and approximate matching. A concept language specifies in an abstract way sequences of design concepts.

The concept language contains:

- *abstract statements* that correspond to one or more statement types in the source code language
- *abstract statement descriptions* that contain the feature vector data used for matching purposes
- *wild character statements +-Satetement and *-Statement* for specifying partial matching, (e.g. abstract statements that match zero or more code statements). The *+-Statement* represents a possible match with one more code statements, while the **-Statement* represents a match with zero or more code statements.
- *typed variables* for dynamically linking variables from pattern fragments to code fragments.
- *operators*: { ||, ⊕, ; } for specifying the order of the matching process. The (||) operator indicates statements that can be interleaved (i.e. share no data dependencies). The (⊕) operator is used to indicate alternative choices while the *(;)* operator indicates sequencing.

A typical example of a pattern code is given below:

```
{
  Iterative-Statement(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" ]
        }
}
```

Here the query or pattern searches for an `Iterative-Statement` (e.g. a `while`, a `for`, or a `do-while` loop) that has in its condition an expression that uses variable `?x` that is a pointer to the abstract type `list` (e.g. array, linked list) and the expression contains the keyword "NULL". The body of `Iterative-Statement` contains a sequence of one or more statements (`+-Statement`) that in its body uses at least variable `?y` (which binds to the variable `obj` in the code match below) and contains the keyword `member`. The `Assignment-Statement` uses at least variable `?x`, defines variable `?x` which in this example binds to variable `field`, and contains the keyword `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;
    }
}
```

5 Concept-to-Code Distance Calculation

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

$$P_r(T_c|T_a) = P_r(r_{c_1},,...r_{c_i},...r_{c_I}|r_{a_1},...r_{a_j},...r_{a_J})(1)$$

where,

$(r_{c_1},...r_{c_i},...r_{c_I})$ is the sequence of rewriting rules used for generating T_c and $(r_{a_1},...r_{a_j},...r_{a_J})$ is the sequence of rules used for generating T_a. An approximation of the (1) is obtained by replacing a rule with its left-hand side symbol.

Let $S_{i_1},..S_{i_k}$ be a string of program statements (represented as AST objects in the local workspace) corresponding to the yield of the non-terminal symbol c_i in T_c. Similarly, let $A_{j_1},..A_{j_n}$ be a string of concept descriptions (represented as AST objects in

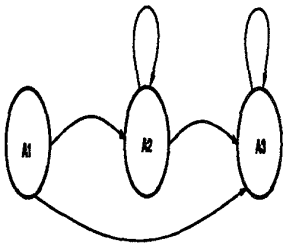

Figure 3: A dynamic model for the pattern $A1; A2^*; A3^*$

the local workspace) corresponding to the yield of the non-terminal symbol a_j in T_a.

The matching process is guided by two categories of automata. The first type is dynamic as it is built at run time while parsing the abstract code description. Nodes represent abstract statements and arcs represent allowable transitions between the abstract statements. The dynamic model is a network of connected models of the second type.

The second type, a Markov model, is static and represents the probability a particular abstract statement generates a code statement. There are as many static Markov models as the abstract statement types in the abstract language.

The sequence of abstract descriptions A_j is used to build dynamically the first type Markov model an example of which is given in Fig.3 This model is built at run time when A_j is found in the conceptual representation. It is constructed from the given pattern and is called Abstract Pattern Model (APM). Each transition of the APM is linked to a static, permanently available Markov model called Source Code Model (SCM).

The best alignment between S_i and A_j may be computed by the Viterbi [16] dynamic programming algorithm using the SCM for evaluating the following probability:

$$P(S_1, S_2, ...S_k|A_j)$$

The probability $P(S_1, S_2, ...S_k|A_j)$ is interpreted as "The probability that the code fragment represented by the sequence of statements $S_1; S_2; ...S_k$ can be generated (matched) by the abstract statement state A_j of the APM automaton".

For exampie given the APM of Fig.3 the following probabilities are computed:

$$P(S_1|A_1) = 1.0$$

$$P(S_1, S_2|A_2) = P(S_1|A_1) \cdot P(S_2|A_2)$$

$$P(S_1, S_2|A_3) = P(S_1|A_1) \cdot P(S_2|A_3)$$

$$P(S_1, S_2, S_3|A_3) =$$

$$Max \begin{cases} P(S_1, S_2|A_2) \cdot P(S_3|A_3) \\ P(S_1, S_2|A_3) \cdot P(S_3|A_3) \end{cases}$$

$$P(S_1, S_2, S_3|A_2) = P(S_1, S_2|A_2) \cdot P(S_3|A_2)$$

Note that when two program statements have already tried, two transitions have been consumed (two matches have been tried with entities in the source code) and the reachable active states are A2 or A3.

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

This static model SCM is used to calculate the $P(S_i|A_j)$ probabilities. On the other hand the dynamic model is created at run-time while parsing the abstract pattern (section 3). With each transition we associate a list of probabilities based on the type of the expression likely to be found as the next expression to be matched 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 involves an **`inequality`** of the form *(list-node-ptr != NULL)* which contains an identifier reference and the keyword NULL.

An example of a static model for the expression-pattern is given in Fig. 4. where A_j is a **`Pattern-Expression`** abstract statement, and S_i is an expression in the source code.

The initial probabilities in the static model are provided by the user who either may gives 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. For example a *traversal of linked-list* plan usually is implemented with a **`while`** loop. The **`Iterative`** abstract statement can be considered to generate a **`while`** statement with higher probability than a **`for`** statement. 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 [8].

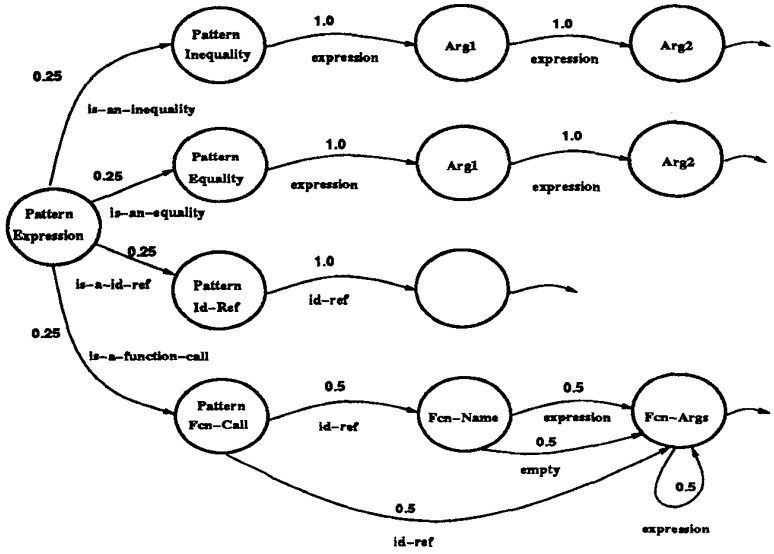

Figure 4: The static model for the expression-pattern. Different models may exist for different plans. For example the *traversal of linked-list* plan may have higher probability attached to the *is-an-inequality* transition as the programmer expects an inequality expression of the form *(field != NULL)* in the condition of the iterative statement.

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:

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

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.

Finally, 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 can be generated by the first pattern statement *b)* keywords and literal variable names should match *c)* metric values (if provided) should be within the given threshold values

Once a candidate list of code fragments has been chosen the actual pattern matching takes place in order to select only these code fragments that best fit the pattern.

6 Conclusion

Pattern matching plays an important role for plan recognition and design recovery. In this paper we have presented two types of pattern matching techniques that are used for code-to-code and concept-to-code matching. Code-to-code matching is used for clone detection and for computing similarity distances between two code fragments. It is based on an dynamic programming pattern matcher that computes the best alignment between two code fragments in terms of insertion, deletion, and substitution costs between statements of a model code fragment and an input code fragment. We have experimented with different code features for comparing code statements. These include, metric values, data and control flow program attributes, keywords, and lexicographical distances between variable names. Using this technique, we are able to detect clones in large software systems $\geq$ 300 KLOC. Clone detection is used to identify "conceptually" related operations in the source code, and identify redundancy in large systems.

Concept-to-code matching is a more ambitious objective and uses an abstract language 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 term of the probability that the abstract statement can generate the particular code statement. Program features that are used to compute similarity are based on statement type, keywords, variable names, data types, metrics, and bindings between abstract variables and variables in the source code. Concept-to-code matching is under evolution using the REFINE environment to support plan localization in C programs.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] C-Language Integrated Production System *User's Manual* NASA Software Technology Division, Johnson Space Center, Houston, TX.

[6] Engberts, A., Kozaczynski, W., Ning, J., "Automating Software Maintenance by Concept Recognition Based Program Transform ation," *In CSM'91 : Proceedings of the 1991 Conference on Software Maintenance*, Oc tober 1991.

[7] Fenton, E. "Software metrics: a rigorous approach", Chapman and Hall, 1991.

[8] Kuhn, R., DeMori, R., "A Cache-Based Natural Language Model for Speech Recognition", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 12, N o.6, June 1990, pp. 570-583.

[9] Kontogiannis, K., DeMori, R., Bernstein, M., Merlo, E., "Localization of Design Concepts in Legacy Systems", *In Proceedings of International Conference on Software Maintenance 1994*, S eptember 1994, Victoria, BC. Canada, pp. 414-423.

[10] Moller, K., Software metrics: a practitioner's guide to improved product development"

[11] Muller, H., Corrie, B., Tilley, S., *Spatial and Visual Representations of Software Structures*, Tech. Rep. TR-74. 086, IBM Canada Ltd. April 1992.

[12] 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.

[13] 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.

[14] Rich, C. and Wills, L.M., "Recognizing a Program's Design: A Graph-Parsing Approach," *IEEE Software*, Jan 1990, pp. 82 - 89.

[15] 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.

[16] Viterbi, A.J, "Error Bounds for Convolutional Codes and an Asymptotic Optimum Decoding Algorithm", *IEEE Trans. Information Theory*, 13(2) 1967.