

A New Multi-level Information Retrieval Technique for Reuse Software Components

Mahmoud M. El-Khouly & Behrouz H. Far & Zenya Koono

Dept. of Information & Computer Sciences, Faculty of Engineering, Saitama University

255 Urawa-shi, Shimo-okubo, Saitama 338 Japan.

{elkhoully, far, koono}@cit.ics.saitama-u.ac.jp

ABSTRACT

In automatic programming area, the users still fail to find available software components that match their needs faster than developing them again. In this paper, a new information retrieval technique to help in retrieving the software components from repositories based on different levels of accuracy (i.e., exact match, match, and similar) had been presented. The software components had been classified into classes according to their functionality, and then they had been stored with what we call 'data dictionary' in a repository. The exact match is not required to retrieve a required software component from this repository using our retrieving technique. An example of how to apply this technique in automatic programming area to produce a program code list automatically had been given. The feature of learning the structure of new software component had been existed.

1. INTRODUCTION

The failure of component retrieving is mainly caused by the disaccord of component designers and other agents who want to reuse it. Each component has the corresponding specification (e.g., name, class, definition, ...). Usually, the agents retrieve the appropriate components according to their names or functionality. However, if the keyword of the component at repository and that of the agent disaccord, the retrieval can be failure even if the eligible component exists in the component repository.

There are many definitions of what software reuse means, but most involve some variation of the *use of engineering knowledge or artifacts from existing systems to build new ones* [1]. This definition encompasses many diverse areas, from application generators, to domain analysis, to design patterns, to classification and search, to management [2].

Our interest is in the use of existing software components, in particular *functions*.

Research dealing with searching software libraries has principally focused on improving indexing [3,4,5]. Others have much attention towards automated methods for gathering information in response to a query from a user [6,7,8,9]. Moreover, most information retrieval systems use Boolean operations for searching large document and collections. While Boolean operations for information retrieval systems [10] have been criticized, improving their retrieval effectiveness has been difficult [11]. Intelligent matching strategies for information retrieval often use concept analysis requiring semantic calculations at different levels [12]. Ambrosio [13] used two auxiliary dictionaries (*Domain* dictionary defining the relationships between different application and *Term* dictionary defining the semantic and syntactic relationships between concepts) to fulfill the previous shortage. However, using auxiliary dictionaries requires a new query process at those dictionaries. In our repository, we included the data dictionary in the components repository such that no new query process is required to search about the semantic of components [14].

In our research, first we classify the software components into classes according to their functionality, and then we store those components with *data dictionary* in a repository. Second, we present a multi-level information retrieval technique contains three levels for retrieving the software component according to its specification (name, similar function, ...) and we use a frame based representation to inherit the supper-class characteristics.

In next section, we present our data item. In section (3) the retrieval system mechanism has been presented. Section (4) gives an example of using this mechanism in automatic programming area, while, Section (5) presents the conclusion.

2. DATA ITEM

An item in our repository is a "class", in the object-oriented sense. A class consists of a set of "methods" which define its functionality. Each method has a set of "instance variables", "operations used", "formula", "similar names", "similar function" and "definition". At the "operations used" (A,M,L,D,P,R,S) represent (addition, subtraction, multiplication, division, power, square root and summation) respectively. The "formula" uses the "operations used" of its method to represent the method's formula. "Similar names" contains the list of the synonym name(s) of the function, e.g. "cos" is the synonym name of "cosine". While "similar functions" contains the list of the function name(s), which can be adapted to behave, like

the original function, e.g. "sin" function is similar to "cos" function. By this structure, we found that the retrieval process becomes faster than using auxiliary dictionary, the contents of the repository become complete, understandable, clear and readable.

2.1 Item Representation

Suppose that we have a first order definition of data item L with signature

$$L = \langle C, N, I, O, R, S, J, M \rangle \quad (1)$$

Where C is a super class; N is a name of the function; I is a set of instance variables; O is a set of operations used; R is a formula of the function; S is a set of synonym functions' names; J is a set of similar functions that may have different names and M is a definition of the function.

We are interested in some formal criteria for obtaining a software component, e.g., a function F, with signature

$$F = \langle N, I, O, R, M \rangle \quad (2)$$

I.e., a function F should retrieve with its name, instant variables, formula and definition.

We use equation (1) in building our repository, while we use equation (2) as what we expect from the retrieving process.

3. RETRIEVAL SYSTEM MECHANISM

In this retrieval system, there are three levels of accuracy, *exact match*, *match* and *similar*. "Find" and "Similar" functions for search had been built. "Find" function is responsible to retrieve *exact match* and *match* software components. While "Similar" function is responsible for retrieving *similar* software components. The query process uses the "Find" function to search at the name of functions in the repository about *exact match* software components. If no satisfactory component is found, the query process changes the order of parameters (to search at the synonym function names) to find *match* software components. If still no satisfactory result is found, the answer returned to the agent is *null*. At that time, the query process replaces "Find" function with "Similar" function (as shown in Figure 1), which uses the data dictionary to find similar software components. The data dictionary contains "instance variables", "operations used", "formula", "similar function" and "definition" for each method in the repository. Retrieving its data dictionary's information expands the selected method. If it needs to be adapted, we consult the table of variable names (as described in section 4.2) to adapt it according to the variable names required.

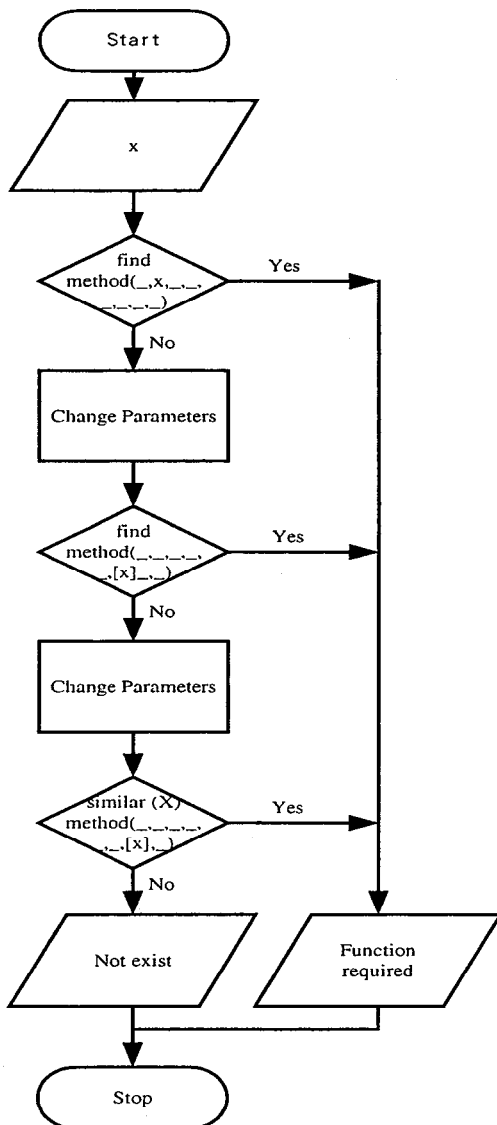


Figure (1): Retrieval system mechanism

3.1 Find and Similar

Def.1 A function F has *exact match* with L iff

$$(F(N)=L(N)) \quad (3)$$

It means that a function F exactly *match* with the data item in the repository if and only if they have the same name.

- Rule 1:** If it is the first node
 then 1) write the name of the node + SEQ
 2) infer all other nodes
 3) write the name of the node + END
- Rule 2:** If the node has no child & no SELECT NOTATION
 then put do before its name.
- Rule 3:** If the node has one or more child and don't contain (LOOP or SELECT notations)
 then 1) write the name of the node + SEQ
 2) put do before the name of each child
 3) write the name of the node + END
- Rule 4:** If the node has LOOP notation
 then 1) write the name of the node + ITR UNTIL + node CONDITION
 2) infer all children
 3) write the name of the node + END
- Rule 5:** If the node has SELECT notation
 then 1) write SEL + the name of the node + YES
 2) infer positive branch
 3) write ALT + the name of the node + NO
 4) infer negative branch
 6) write END

Figure (2): Rules in Converter function.

Def.2 A function F *matches* with L iff

$$(F(N) \in (L(S))) \quad (4)$$

It means that the function F *matches* with the data item in the repository if and only if the name of the required function exists in the list of similar names of that data item.

Def.3 A function F is *similar* to L iff

$$((F(N) \in (L(J))) \quad (5)$$

I.e., the function F is similar to the data item in the repository if and only if the name of F exists in the list of similar function names of that data item.

Find function uses definitions 1 and 2 to retrieve *exact match* and *match* software components, while **Similar** function applies definition 3 to find similar software components.

4. AUTOMATIC PROGRAMMING

To show the efficiency for the above mentioned retrieval technique, we present an example about building a source code automatically given the input and output data in the form of Jackson System Processes (JSP). The system can produce a pseudo code list for the required program. First, the system consults the repository to extract the required functions in order to construct a complete program data structure. Second, the function "Converter" (see Figure 2) will be invoked to transform the program data structure to the pseudo code list. However, converter functions for other programming languages can be build. In the following sub-section, we introduce a brief introduction about JSP. Then an example of producing program automatically is obtained.

4.1 Jackson System Process

Jackson System Process (JSP) [15] is a method of programming design which has arisen within the field of commercial Data Processing (DP) as a part of a whole new approach to systems development [16]. JSP is a prescriptive design method, which can be taught and can be learned. It enables a trainee programmer to learn how to design programs without resort to trial and error or the 'sit by Fred' approach, in which he/she is asked to sit next to an experienced programmer until by some magical process he/she assimilates the latter's knowledge. Using JSP, different programmers will produce similar programs, which makes it easier for one programmer to understand and maintain another's. Although JSP has shortcoming in real-time applications, but we found it very suitable in this application. Moreover, the theory of JSP cannot tell us how to code a program, but in practice the implications of JSP for coding are immense, such that, we put rules to transform the JSP program data structure to any programming language's source code.

JSP has four component types. They are: elementary components, which are not further dissected and have no parts, and three composite types: (a) sequence, which has two or more parts occurring once each, in order, (b) iteration, which has one part, occurring zero or more times, and (c) selection, which has two or more parts of which one, and only one, occurs once. The three composite types form an effective structuring system. Each is easy to understand, in the sense that the relationship between its parts corresponds to an intuitively acceptable idea, which is simple and easy to remember. Also, it has been shown that any program which can be expressed in a flowchart can be expressed as a structure of sequences, iterations and selections, so there is no restriction on the programs we may write [15].

We used Jackson System Process as a systematic way of design, since dealing with a large and complex program, we need some way of tackling the work that allows us to devote our attention to only a small part of it at any time. Hierarchical structuring has the same way of design.

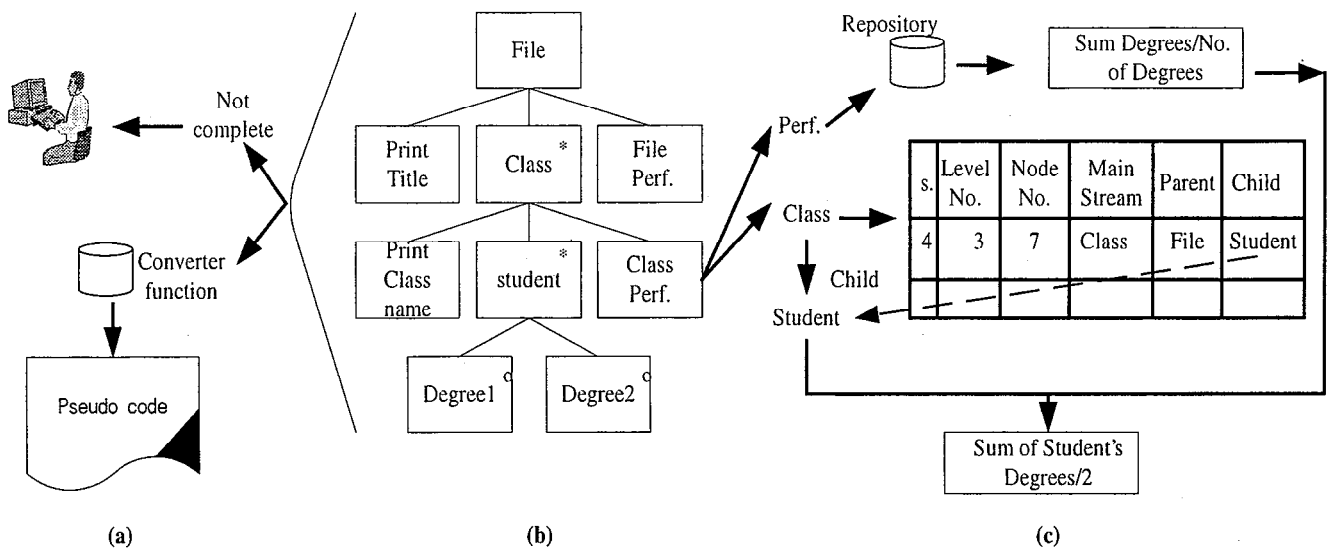


Figure (3): Producing Program Automatically

Hierarchical structuring is powerful because it allows us to create large and complex programs without using large and complex components.

4.2 Example

We give here an example for Producing Program Automatically (PPA). Consider a student file which contains the name of students, their numbers, their degrees, etc. and we simply want to print out each class with its performance, and the total file performance at the end. So, the manager of the school (the user) will give to the trainee programmer just the input data file (which contains, student number, student name, class name and student's degrees), and the output list required (which contains some fields do not appear at the input data file e.g., class performance). The programmer will use JSP notations to design program data structure (as shown in Figure (3-b)). Then, he/she supplies it to the system in the form of two dimensional array "node[row, column]" in which "node[1,1]" means the first node in the chart and "node[2,3]" means the third node at the second level in the chart. The system scan those nodes (except those on a mainstream of the JSP graph) to check whether they contain components which are known to the system or not?. If not, then the system acts according to the following algorithm:-

- 1- break the node's name (if necessary) to variable names.
- 2- if one of those names is related to the mainstream of the JSP graph, then it will be added to the variable names table which contains (level number, node number, the name of the mainstream, the node's parent, and the child's name).
- 3- if not, then the system consult the software components repository using "Find" and "Similar" functions to find the formula for the required function.

- 4- the system adapts this formula (if found) using child's name of the node (as shown in Figure 3-c).
- 5- step 1 to step 4 are repeated until scanning all nodes.

The result of the above algorithm is a complete program data structure, which then will be converted to the pseudo code list using the "Converter" function (as shown in Figure 3-a). However, if there is incomplete program data structure, the system consults the user to complete it. At this point, we are developing now a multi-agent-based system, which will consult other software agent's repositories to complete the program data structure instead to consult the user.

4.3 Implementation

We implemented the system using PROLOG, since it is especially well suited for problems that involve objects – in particular, structured objects- and relations between them. The software components had been stored as clauses, while the search technique had been added at the beginning as predicate rules. By this way, we can add/modify the software components as well as the predicate rules. Also, the feature of learning the structure of new software components exists. This will be done when the "Find" function fails to get exact match or match software components, so the result of "Similar" function and the contribution from the user, will be added to the above clauses, such that in the new invoke, it will be taken into consideration. Figure (4) depicts the accumulated number of rules vs. number of invokes. As can be seen, the accumulation curve shows that rapid increase at first, then the rate decreases gradually though the increase continues. This resembles a learning effect. At the flat area of the curve, using already extracted methods makes full automatic programming. Thus, after enough rules are learned, automatic programming is possible for additional

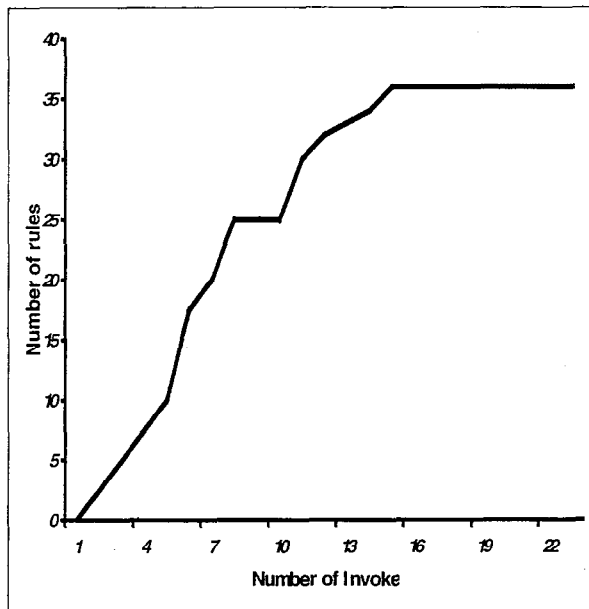


Figure (4): Learning effect

features.

5. CONCLUSIONS

The effectiveness of the reuse based software development is strongly dependent on the underlying classification scheme and retrieval techniques. In this paper, we tried to cover both. We began from constructing the repository and we fulfill its shortages by supporting it with a data dictionary. We present a retrieval technique with different levels of accuracy. The advantage of this technique is that the exact keywords match is not necessary. We also presented an application in automatic programming area. This technique has a potential to be applied in multi-agent environment.

6. REFERENCES

- [1] W. Frakes and S. Isoda, "Success factors for systematic reuse," *IEEE Software*, Sept. 1994, pp 14-22.
- [2] T.J. Biggerstaff and A. J. Prelis, eds., "Software Reusability," ACM Press, Vol. 1, New York 1989.
- [3] S.D. Fraser, J.M. Duran and R. Aubin, "Software Indexing For Reuse," *Proc. 1989 IEEE International Conference On Systems, Man and Cybernetics*, 1989, pp 853-858.
- [4] R. Prieto-Diaz, "Implementing Faceted Classification For Software Reuse," *CACM*, Vol 34, 1991, pp 89-97.
- [5] Y.S. Maarek, D.M. Berry and G.E. Kaiser, "An Information Retrieval Approach For Automatically Constructing Software Libraries", *IEEE Transactions On Software Engineering*, Vol. 17, No. 8 Aug., 1991, pp 800-813.
- [6] Y. Arens, C. Y. Chee, C. Hsu, and C. A. Knoblock, "Retrieving and integrating data from multiple information sources," *International Journal on Intelligent and Cooperative Information Systems*, 2(2), 1993, pp.127-158.
- [7] M.C. Bowman, P.B. Danzig, U. Manber, and M.F. Schwartz, "Scalable Internet Resource Discovery: Research Problems and Approaches," *Communication of the ACM*, 37(8), 1994, pp 98-107.
- [8] Tim Oates, M.V. Negendra Prasad and V.R. Lesser, "Cooperative Information Gathering: A Distributed Problem Solving Approach," Technical Report 94-66, Dept. Of Computer Science University of Massachusetts, Amherst, 1994.
- [9] Shigeru Fujita, Hideki Hara, Kenji Sugawara, Tetsuo Kinoshita and Norio Shiratori, "Agent-Based Support for Reusing Components in Library," *Knowledge-Based Software Eng.*, P.Navrat and H.Ueno (Eds.) IOS Press, 1998.
- [10] S. Wartik, "Boolean operations," *Info. Retrieval: Data Structures & Algo.* W. B. Frakes, and R. Baeza-Yates (eds), Prentice Hall 1992, PP 264-292.
- [11] W. B. Frakes, "Introduction to information storage and retrieval systems," *Info. Retrieval: Data Structures & Algo.* W. B. Frakes and R. Baeza-Yates (eds), Prentice Hall 1992, PP 1-27.
- [12] Kaname Funakoshi and Tu Bao Ho. "A Rough Set Approach to Information Retrieval," *Studies in Fuzziness and Soft Computing - Rough Sets in Knowledge Discover*, Lech Polkowski & Andrzej Skowron (eds), Physica-Verlag Heidelberg, 1998.
- [13] A. P. Ambrosio, "Introducing Semantics in Conceptual Schema Reuse," *CIKM'94, Proceeding of the 3rd Int. Conf. On Info. & Know. Management*, 1994, PP 50-56.
- [14] Mahmoud El-Khouly, Behrouz Far, Aboul-Ella Hassanine and Zenya Koono, "Reuse software components in multi-agent environment," *Proceeding of 33rd Conf. on Statistics, Comp. Sci. and Operations Researches*, Dec. 12-14 1998.
- [15] M. A. Jackson "Principles of Program Design," Academic Press, 1975.
- [16] Ralph Storer, "Practical Program Development using JSP," Blackwell Sci. Pub., 1987.