



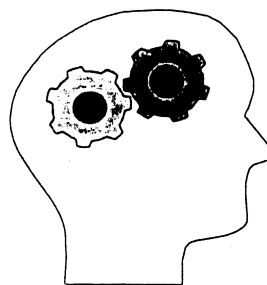
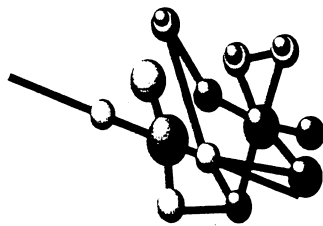
Egyptian Computer Society



The American University in Cairo

PROCEEDINGS OF  
THE SEVENTH INTERNATIONAL  
CONFERENCE  
ON ARTIFICIAL INTELLIGENCE  
APPLICATIONS

CAIRO, EGYPT, FEBRUARY 3 - FEBRUARY 7, 1999  
CAIRO INTERNATIONAL CONFERENCE CENTER  
<http://www.cs.aucegypt.edu/aiconf/>



7<sup>th</sup> ICAIA '99

VOLUME I

# Software-Agent for Reuse Software Components

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

Computer Sciences & Information Dept. Saitama University, Japan.  
{elkhouly, far, koono}@cit.ics.Saitama-u.ac.jp

## Abstract

Many important and useful applications for software agents require multiple agents on a network that communicate with each other. Such agents can retrieve software components for reuse from their repositories. However, if the keywords of the retrieving disaccord, the retrieval can be failure even if the eligible components exist in the software components repository. We propose an information retrieval technique to help in retrieving the software components that an agent wants to use from other agents' software components repositories. This technique uses semantic check to compare between the retrieved component with the agent requirement. The advantage of this technique is that the exact match is not necessary to find a similar software component.

## 1. Introduction

Research dealing with searching software libraries has principally focused on improving indexing [1,2,3]. Others have much attention towards automated methods for gathering information in response to a query from a user [4,5,6,7]. In our research, first we classify the software components into classes according to its functionality and store it in the repository. Second, we establish a new model contains two levels for retrieving components from a repository. In the first level, we present three approaches for retrieving the software components according to its specifications (name, similar function, ...) and we use a frame based representation to inherit the super-class characteristics. In the second level, we use a semantic check for the retrieved component with the definition required. We give an example of how to apply the above steps in automatic programming area.

Figure 1, shows the overall system mechanism applying in a multi-agent environment. The user specifies his input data, output required and data specification for the terms he uses (Figure 1-a). A programmer generates only two data structures (one for input and the other for output) (Figure 1-b). By supplying programmer's output to the agent, the agent produces program data structure after retrieving the suitable software components from the repository. Then, the agent generates the source code referring to the converter rules base [8] (Figure 1-c). If the user asks from beginning for semantic check, then the programmer supplies also the data specification table to be considered during retrieving process.

In the next section, we introduce the background, which includes repository structure and Jackson system process. In Section (3) we present the retrieval system mechanism from constructing the repository until implementation phase, and we give an example in automatic programming at Section (4). Finally, Section (5) presents the conclusion.



## 2. Background

### 2.1 Repository structure

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 the name or functionality. However, if the keyword of the components at repository and that of the agent disaccord, the retrieval can be failure even if the eligible components exist in the component repository.

Moreover, most information retrieval systems use Boolean operations for searching large document, collections. While Boolean operations for information retrieval systems [9] have been criticized, improving their retrieval effectiveness has been difficult [10]. Intelligent matching strategies for information retrieval often use concept analysis requiring semantic calculations at different levels [11]. Ambrosio [12] 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 almost of the contents of that dictionaries in our repository, such that no new query process is required to search about the semantic of components.

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”, “formula”, “similar names”, “similar function” and “definition”, as shown in Figure 2. By this

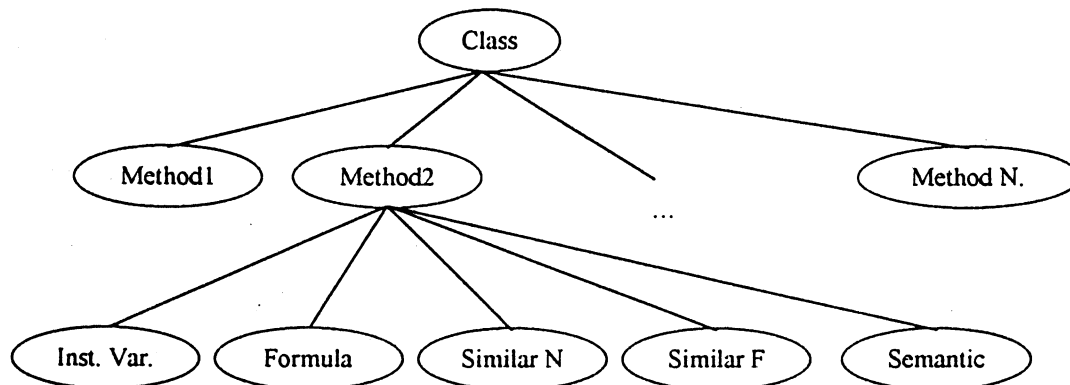


Figure 2: Class hierarchy

structure, we found that the retrieval process becomes faster than using auxiliary dictionary, the contents of the repository become complete, understandable, clear and readable.

## 2.2 Jackson System Process

Jackson Structured Programming (JSP) [13] is a method of program design which has arisen within the field of commercial Data Processing (DP) as part of a whole new approach to systems development [14].

Using JSP, different programmers will produce similar programs, which makes it easier for one programmer to understand and maintain another's. That is the reason why we selected JSP method rather than other object oriented approaches. 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' source code.

## 3. Retrieval System

### 3.1 Data item

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

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

where C is a super class; N is a name of the function; I is a set of instance variables; R is a formula of the function; S is a set of similar functions' names of the function; J is a set of similar functions' function of the function and M is a semantic of the function.

We are interested in some formal criteria for obtaining a function F, with signature

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

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.

### 3.2 Find & Similar

**Def.1** A function F is Exact match with L iff

$$((L(N) = F(N)) \vee (F(N) \in L(S))) \wedge ((F(M) \subseteq L(M)) \vee (L(M) \subseteq F(M)))$$

It means that a function F is exact match with the data item in the repository if and only if the following two conditions are true: (a) they have the same name; or the name of the required function exists in the list of similar names of that data item. (b) The semantic of the required function is subset or equal to the semantic of that data item; or vice versa.

**Def.2** A function F is Match with L iff

$$(L(N) = F(N)) \vee (F(N) \in L(S))$$

It means that the function F is match with the data item in the repository if and only if they have the same name; or 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 if

$$(L(C) = F(C)) \wedge (F(N) \in L(J))$$

I.e., the function F is similar with the data item in the repository if and only if their super class has the same name and the name of F exists in the list of similar functions of that data item.

We use the above definitions to construct two important commands in our retrieval system:

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

### 3.3 Query process:

The interaction between the agent and the repository of reusable software components must follow a well-defined process. This process will control both the visual layer and the access to the repository.

**Def.4** Query process system can be formulated as a quadruple

$$S = (\alpha, \beta, \varphi, \lambda) \quad (3)$$

Where  $\alpha = \{\alpha_1, \alpha_2, \dots, \alpha_M\}$  is a set of index terms (e.g., keywords);  $\beta = \{\beta_1, \beta_2, \dots, \beta_N\}$  is a set of clauses in the repository,  $\beta_j$  has the form of (1), and each  $\beta_j \subseteq \alpha$ ;  $\varphi = \{\varphi_1, \varphi_2, \dots, \varphi_L\}$  is a set of queries each  $\varphi_p \subseteq \alpha$ ; and  $\lambda: \varphi \times \beta \rightarrow \mathfrak{R}^+$  is a ranking function that evaluates the relevance between a query and a clause. In a general form a clause  $\beta_i$  can be denoted as a set of index term-weight pairs

$$\beta_j = (\alpha_{j1}, \omega_{j1}, \alpha_{j2}, \omega_{j2}, \dots, \alpha_{jn}, \omega_{jn}) \quad (4)$$

where  $\alpha_{jr} \in \alpha$  and  $\omega_{jr} \in [0,1]$ ,  $r = 1, 2, \dots, n$ , reflect the relative importance of terms  $\alpha_{jr} \in \beta_j$ . A query  $Q \in \varphi$  can also be denoted as a set of index term-weight pairs

$$Q = (\alpha_{q1}, \omega_{q1}, \alpha_{q2}, \omega_{q2}, \dots, \alpha_{qm}, \omega_{qm}) \quad (5)$$

where  $\alpha_{qs} \in \alpha$  and  $\omega_{qs} \in [0,1]$ ,  $s = 1, 2, \dots, n$ . The query process task is to yield a set  $A = \{l_{a1}, l_{a2}, \dots, l_{an}\} \subseteq \beta$  to the query  $Q$  with a ranking order of  $\lambda(Q, l_{ai})$ . We used these term-weight pairs in semantic check-level.

The element of the retrieval mechanism responsible for the automatic modification of queries is called the Query Process. The query process permits the agent to choose one of three approaches. In the first, components are retrieved only if an exact match is found. This is the traditional approach to database querying. In the second, query modification is applied if no exact match could be found. In this case, the query modification tries to find similar names. In the third approach, query modification is applied again to find a name of similar function, which performs the same action.

### 3.4 Query modification

As we have seen, the query process permits the definition of exact and imprecise queries. The first is treated in the traditional way, by interrogating the repository to find components that satisfy

the conditions specified. At this stage the query process uses the command Find to search in the repository. If no satisfactory component is found, the query process uses the second type by changing the order variables in the Find command to search about similar names. If still no satisfactory component is found, the answer returned to the agent is null. For the third type, the query process replaces (Find command) with (Similar command) which tries to find similar functions' names. Therefore, if a query states that it wants all components that have for example, "cosine" as a key concept, the query will access the repository and recover all the synonyms of "cosine" (e.g., "cos") in the radius stated in the query.

In this approach, if the retrieving request does not contain semantic check, then the agent scans the methods names from top to bottom and selects the first one that seems of sufficient interest. But, if the retrieving request contains semantic check, then we extract those methods which seem of sufficient interest, and then apply a method (see next section) to check its semantic, to find a most suitable method for accurate retrieval.

### *3.5 Semantic check:*

To use a sophisticated natural language processing techniques to check the semantic of the retrieved software component with that one required by the agent, will not lead to satisfied performance due to the different definitions used by different agents. The program called DowQuest [15] is a commercially available system with a basic command line interface, but very sophisticated functionality. It allows users to type in a sentence (e.g. 'Tell me about the eruption of the Alaskan volcano'), get a list of articles. DowQuest does not try to interpret the meaning of the question; in the above sentence, the system will drop out the words "tell", "me", "about", "the", and "of" and use the lower frequency words to search the database. The system however, appeared to understand plain English; but in reality it made no effort to understand the question that was typed in -it just used a statistical algorithm [16].

The way that DowQuest work is exactly what we need at our semantic check phase, since we don't want to sink at natural language processing to compare two sentences, since it will take a time and will not lead to satisfied results.

We check our repository and found that, there are words like "the", "of", "for", "an", "in", "a" can be drop out during semantic check. Therefore, after extracting those components, which seem of sufficient interest from the repository, we drop out the above words from their semantic and then make some statistical evaluation to obtain the most suitable one for the definition required.

### *3.6 Implementation*

We implemented the system using IF/Prolog Version 5.0B, since Prolog is especially well suited for problems that involve objects- in particular, structured objects- and relations between them. The software components were stored as clauses, while the search technique had added at the beginning as predicate rules. By this way, we can add/modify the software components as will as the predicate rules. Also, the feature of learning the structure of new components is existing.

## **4. Automatic Programming Example**

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 (Figure 1-a).

## a) Assumptions:

We consider the following assumption for agent A:

- Agent A is responsible for making pseudo code program for this problem, given input and output data structures in form of JSP.
- Agent A searches within its repository about similar cases that match the current problem, in order to construct the program data structure.
- If agent A cannot construct program data structure successfully, it needs to know the function required to transfer that input to the required output, at that time agent A begins to negotiate with other agents using Knowledge and Query Manipulation Language (KQML) [17].
- After constructing program data structure, agent A uses converter rule base (for pseudo code) to produce the source program.

We consider the following assumption for other agents (for simplicity, we'll use two agents called agent B and agent C):

- Agents B and C are software houses dealer which sell software modules or functions to customers;
- They want to gain more customers. They have always a secondary goal, which is to make a customer satisfied. That will done by selling the only suitable module/function to the customers, and to apologize if the required function is not available at that time, or advice to purchase it from another agent(s);
- The profit here is the customers' satisfaction.

## b) Approach:

First: we will provide input data structure and the output data structure to agent A.

Second: Agent A will search in his repository about the suitable functions which required transforming the input data to the output data.

Third: If agent A doesn't complete his target, from his own repository then, it negotiates with other agents.

After constructing program data structure, the agent uses our built converter rules to transform JSP program data structure to pseudo code. We used KQML to communicate with other agents rather than using other languages, e.g. CORBA [18], because we need to adapt the method's formula in our future research to be suitable for any number of variables needed by the caller agents. As the result of this approach, we found that by supplying agent A with only two charts (in the above example, they contain 12 nodes) which is transformed to (59 lines in pseudo code without comment lines) and (74 lines in C without comment lines). Considering that the time required for producing one code is equal to the time required for producing one of chart's nodes, then we found that we save 79.6%~83.7% of the original time required.

## 5. Conclusion

The effectiveness of the reuse-based approach to software development is strongly dependent on the underlying classification scheme and retrieval mechanism. In this paper we tried to cover both, by enhancing the structure of the repository, and in the retrieving point, we described a new model consists of two level of retrieving, retrieve without semantic check and retrieve with semantic check. For the first level, we presented three approaches to find a software component in the repository which satisfies the rules of (exact match; match; or similar). The advantage of this model is that the exact keywords match is not necessary to find similar components. We also, presented the

application of this new model to automatic programming area. The technique has potential to be applied to computer tutoring systems.

## References

- [1] S.D. Fraser, J.M. Duran and R. Aubin, "Software Indexing For Reuse," Proc. 1989 IEEE International Conference On Systems, Man and Cybernetics, pp 853-858, 1989.
- [2] R. Prieto-Diaz, "Implementing Faceted Classification For Software Reuse," CACM, Vol. 34, pp. 89-97, 1991.
- [3] Y.S. Maarek, D.M. Berry & G.E. Kaiser, "An Information Retrieval Approach For Automatically Constructing Software Libraries," IEEE Transactions On Software Engineering, Vol. 17, No. 8, pp 800-813, Aug. 1991.
- [4] Y. Arens, C.Y. Chee, C. Hsu, and C.A. Knoblock, "Retrieving and integrating data from multiple information sources", in International Journal on Intelligent and Cooperative Information Systems, 2(2), pp.127-158, 1993.
- [5] 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), pp. 98-107, 1994.
- [6] T. Qates, M.V. Negendra Prasad, V.R. Lesser, "Cooperative Information Gathering: A Distributed Problem Solving Approach," Technical Report 94-66, Dept. Of Computer Science University of Massachusetts, Amherst, 1994.
- [7] S. Fujita, H. Hara, K. Sugawara, T. Kinoshita and N. Shiratori, "Agent-Based Support for Reusing Components in Library," Knowledge-Based Software Eng., P. Navrat and H. Ueno (Eds.), IOS Press, 1998.
- [8] M. El-Khouly, B.H. Far, A.E. Hassanine, and Z. Koono, "Reuse software components in multi-agent environment," Proceeding of 33rd Conf. On Statistics, Comp. Sci. and Operations Researches, December, 1998.
- [9] S. Wartik, "Boolean operations", In Frakes, W.B. and Baeza-Yates, R.(eds), Info. Retrieval: Data Structures & Algo., Prentice Hall 1992, pp. 264-292.
- [10] W.B. Frakes, "Introduction to information storage and retrieval systems," In Frakes, W.B. and Baeza-Yates, R.(eds), Info. Retrieval: Data Structures & Algo., Prentice Hall 1992, pp. 1-27.
- [11] L. Polkowski and A. Skowron (Editors), "Studies in Fuzziness and Soft Computing - Rough Sets in Knowledge Discovery," Physica-Verlag, 1998.
- [12] A.P. Ambrosio, "Introducing Semantics in Conceptual Schema Reuse," in CIKM'94, Proceeding of the 3rd Int. Conf. On Info. & Know. Management, pp. 50-56.
- [13] M.A. Jackson, "Principles of Program Design," Academic Press, 1975.
- [14] R. Storer, "Practical Program Development using JSP," Blackwell Sci. Pub., 1987.
- [15] Dow Jones, "Dow Jones News/Retrieval User's Guide," Princeton, N.J.: Dow Jones and Company, 1989.
- [16] J. Bradshaw, "Software Agents," American Association for A.I., 1997.
- [17] T. Finin, Y. Labrou, and J. Mayfield, "KQML as an agent communication language," In Jeffery M. Bradshaw, editor, Software Agents, MIT press, 1995.
- [18] R. Orfali, D. Harkey, and J. Edwards, "Instant CORBA," John Wiley & Sons, Inc., 1997.