TR-172

Intelligent Support for Office Work with a Prolog-Based Object-Oriented Programming Language ESP

by
Hideki Sato and Hitoshi Matsumoto
(Fujitsu Ltd.)

April, 1986

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Intelligent Support for Office Work with a Prolog-Based Object-Oriented Programming Language ESP

Hideki Sato and Hitoshi Matsumoto

FUJITSU LIMITED

1015 Kamikodanaka, Nakahara-ku, Kawasaki, 211, Japan

ABSTRACT

This paper discusses intelligent support for office work with ESP, a Prolog-based object-oriented programming language. To provide intelligent support for the office work, the knowledge of the office work must be represented and maintained, and meta-level procedures must be developed for the system to use the object-level of the knowledge to act intelligently and actively. ESP is used to model the knowledge and to develop the meta-level procedures. Both the object-oriented feature and the logic-programming feature of ESP are sufficient to provide intelligent support. Discussion is focused on office tasks modeling and on how office tasks are performed, for office tasks are the basis for making an office system intelligent and active. In our modeling framework, an office task is represented by a class of ESP. The predicate form of ESP is used to represent constituent tasks, pre-conditions, post-conditions, and constraints of the tasks. The meta-level procedures are implemented as method predicates defined in the system classes and inherited by classes representing office tasks for intelligent support.

This work is part of the activities undertaken in the Fifth Generation Computer Systems (FGCS) Project of Japan.

1. INTRODUCTION

As computer technologies progress, various kinds of software tools have been developed to support office work such as electronic mail, calendar management, text editor, and forms packages. Although these tools are effective for performing specific aspects of office tasks, supporting this level of tasks is limited to the effect of improvement on office productivity [12]. To improve office productivity, more effective support is required for higher level tasks directly related to office goals or office functions. Currently, computer systems play only a passive role in performing such tasks, while office workers play an active role. This is because most of the knowledge resides with the office workers, not the computer systems. An intelligent and active office system must be developed, if this situation is to change.

Methods for supporting higher levels of office work can be divided into two categories. One category is based on the procedure execution approach; the other is based on the knowledge-based approach. The procedure execution approach [11],[14],[17] regards office work as a data processing activity: office tasks are structured in terms of input, process, and output. The assumption is that office tasks can be well-understood and completely automated. Therefore, the tasks this approach deals with best are restricted to routine and rigidly structured tasks. This approach seems inadequate for supporting office work, because of the nature of office and the office work [5].

The knowledge-based approach [1],[2],[3],[6],[13],[16] regards office work as a problem-solving activity: office tasks are structured in terms of goal-subgoal hierarchy. The assumption is that not all aspects of office tasks are totally understood and that the knowledge used for problem-solving may only be partial. This approach seems promising, for it enables the system to use various kinds of the stored knowledge about office work to provide a wide range of intelligent support.

Our approach to supporting office work is in line with the knowledge-based one. To create a knowledge-based office system, two things are required. First, the system must be able to represent and maintain knowledge about office work. This includes knowledge about organizations, office workers, data objects, office tasks, responsibilities, and so on. Second, the system must be able to help office workers with their office tasks. The system knows what office tasks are to be done, when they are to be done, and who is responsible for doing them. It can assist in analyzing and monitoring. It performs simple routine tasks for office workers. It helps office workers to perform complex tasks. In a word, the system acts intelligently based on its stored knowledge. Such an intelligent and active system requires that meta-level procedures [10] be added to the knowledge mentioned above. The meta-level procedures refer to object-level knowledge [10].

The object-oriented paradigm [8],[9],[15] and the logic-programming paradigm [7],[10] are adequate for coping with the requirements mentioned above. The first requirement is met by representing concepts and relationships in the office. Concepts can be modeled in a natural way with the object-oriented paradigm. Relationships between concepts can be nicely represented with the logic-programming paradigm. The second requirement is easily met by the use of both paradigms to develop the meta-level procedures that incorporate the above knowledge.

This paper discusses intelligent support for office work with ESP [4], a Prolog-based object-oriented programming language. The following section is a brief explanation of ESP. Section 3 focuses on our framework for office task modeling. Section 4 describes how office tasks are performed, based on the task descriptions. Our conclusions are given in Section 5.

2. ESP

ESP is a Prolog-based object-oriented programming language which provide both an object-oriented feature and a logic-programming feature. An object of ESP is defined in terms of a set of methods implemented by Prolog clauses and a set of slots. ESP programming describes a class definition consisting of a syntax as shown in Figure 1.

```
     class <class name> has
            [ <nature definition> ; ]             ....  inheritance
            { <class slot definition> ; }         ....  class
            { <class clause definition> ; }       ....  class
   [ instance
            { <instance slot definition> ; }      ....  instance
            { <instance clause definition> ; } ]  ....  instance
   [ local
            { <local clause definition> ; } ]     ....  local
     end.
```

Figure 1. Syntax of class definition in ESP

The inheritance part defines the super classes which this class inherits. If a class inherits other classes, the class has all the methods and slots of those classes as well as its own. The class part defines the class object which consists of class methods and class slots. The instance part defines the template of instance objects. Its definition consists of instance methods and instance slots as well as a class definition. A method is defined by Prolog clauses. The local part defines the predicates which can only be called within this class.

A method predicate has the syntax ":<predicate name>(<argument list>)"; It is distinguished from a local predicate by the colon ':' preceding <predicate name>. The beginning element of <argument list> specifies the object which receives the message. Before-demon and after-demon predicates can be attached to a method (primary) predicate. They are specified by putting the keywords 'before' and 'after' in front of the syntax of the method predicate. During execution of a message, before-demons are called before the call of the primary predicate, and after-demons are called after the call of the primary predicate. If a class inherits other classes, the clauses of the same primary method are OR'ed and the demons are AND'ed with the primary predicate.

Some built-in method predicates are provided for object manipulation. A ":new(Class,Instance)" method predicate is used to create an instance object of a class. ":get_slot(Object,Slot_name,Value)" and ":set_slot(Object,Slot_name,Value)" method predicates are provided to access a slot of an object. They are also described in the macro notation, "Object!Slot_name" to refer to the value of the slot and "Object!Slot_name:=Value" to assign the value to the slot.

3. FRAMEWORK OF OFFICE TASK MODELING

In the office, there are various concepts related to office tasks. They are also related to each other. Such concepts are organizations, office workers, data objects (e.g., forms), office tasks, and so on. The most important concept among them is the office task, because it plays a central role in making an office system intelligent and active. This section discusses how to model office tasks. ESP is used as the model representation language.

In our modeling framework, an office task is represented by a class of ESP. From the standpoint of modeling, the super-sub relationship and the whole-part relationship are useful. The former enables the user to easily define the task description similar to existing ones with a few incremental changes. The latter provides the user with a means to compose an office task of another constituent tasks. Three basic abstract classes (i.e., 'task' class, 'composite_task' class, and 'primitive_task' class) are provided. 'task' class is the most abstract concept of all the tasks. 'composite_task' class is the abstract concept of composite-tasks, while 'primitive_task' class is the abstract concept of primitive-tasks. They are both subclasses of 'task' class. A task concepts in the office is defined as a subclass which is linked to the 'composite_task' class or the 'primitive_task' class either directly or indirectly.

Figure 2 shows a typical example of a composite-task description. The example is of an accounting of business trip expense. It consists of filling out a travel_expense_account_form, approval of the account request by a manager, completion of the transaction by a clerk, and making a report that explains why the travel expense exceeds the standard expense.

```
class travel_expense_account_task has
% super task of this task
    nature composite_task;
instance
% attributes of this task
    attribute
      petitioner,
      travel_expense_account_form,
      report_on_overspending;
% pre-condition of this task
    before:execute(Self):-
        exist_travel_order_form(Self!petitioner),
        !;
% post-condition of this task
    after:execute(Self):-
        equal(Self!travel_expense_account_form!status,"checked"),
        !;
%constituent_tasks of this task
    :constituent_task(Self,fill_out_travel_expense_account_form,
                      [petitioner(Self!petitioner)]);
    :constituent_task(Self,make_a_report_on_overspending,
                      [petitioner(Self!petitioner)]):-
        judge_overspending(Self!travel_expense_account_form!
                           total_account);
    :constituent_task(Self,get_seal_of_a_manager,
        [manager(Manager),form(Self!travel_expense_account_form),
         report(Self!report_on_overspending)]):-
        :manager(Self!petitioner,Manager);
    :constituent_task(Self,complete_travel_expense_account,
                      [form(Self!travel_expense_account_form),
                       report(Self!report_on_overspending)]):-
        !;
```

```
% constraint
    :followed_by(Self,fill_out_travel_expense_form,
                     get_seal_of_manager);
    :followed_by(Self,get_seal_of_manager,
                     complete_travel_expense_account);
    :followed_by(Self,fill_out_travel_expense_account_form,
                     make_a_report_on_overspending);
    :followed_by(Self,make_a_report_on_overspending,
                     get_seal_of_manager):-
        !;
    :optional(Self,make_a_report_on_overspending):-
        !;
    :match(Self,travel_expense_account_form,
               [fill_out_travel_expense_account_form,form]);
    :match(Self,report_on_overspending,
               [make_a_report_on_overspending,report]):-
        !;
local
    ..........
end.
```

Figure 2. A composite-task description

A composite-task is defined in terms of attributes, pre-condition, post-condition, constituent tasks, and constraints. The attributes of a task relate the task to other objects or values and are represented by the slots of a class. They may be shared by the constituent tasks and used by higher level subsuming tasks. If an attribute has an initialization specification, it is initialized at the instantiation of a task object. In Figure 2, a petitioner, a travel_expense_account_form, and a report_on_overspending are defined as the attributes of the 'travel_expense_account_task'.

Before a task can begin, certain conditions must meet the state of objects in the system. These are called the pre-condition of a task. Also, when a task is completed, certain conditions must be satisfied. These are called the post-condition of a task and are used to ascertain if the goal (i.e., the task) was attained. These conditions are described in the bodies of the clauses, which define the before-demon predicate and the after-demon predicate of the instance method predicate ":execute(Task)". The pre-condition in Figure 2 specifies that the travel_order_form must be submitted for the petitioner before the travel_expense_account_task begins. The post-condition specifies that the travel_expense_account_form must have been checked when the travel_expense_account_task finishes.

A composite-task is composed of lower level constituent tasks which are either composite-tasks or primitive-tasks. These constituent tasks are regarded as the operators which are used to attain the goal (i.e., the task). They may have conditions, if their application depends on the state of objects in the system. They also have parameters required for their execution. This knowledge is defined by the instance method predicate ":constituent_task(Self,Constituent_task,Parameter_list)", which shows the relationship that this task ('Self') has 'Constituent_task' as a constituent task and that the parameters to be passed to the constituent task are specified by 'Parameter_list'. The arguments 'Constituent_task' and 'Parameter_list' give a name and a parameter list for the constituent task. The parameter is a term of a parameter name and its value. The body of the clause which defines the ":constituent_task" predicate, specifies the condition under which the constituent task is applied and the derivation of any parameters. As for the constituent task 'make_a_report_on_overspending', making a report is required if the travel expense exceeds a standard value and takes the petitioner of the travel expense account as its parameter.

The temporal ordering among the constituent-tasks must be determined to perform a composite-task. However, office tasks are often parallel, negligible, and performed in several ways. Therefore, it is difficult to totally specify the precise sequence of constituent tasks. Instead, the definition of constraints is introduced to specify the relationships among constituent tasks and optional constituent tasks piecemeal. The following three instance method predicates are used to define such constraints.

- :followed(Self,Constituent_task1,Constituent_task2)

  This predicate shows the relationship that 'Constituent_task1' is followed by 'Constituent_task2' during the execution of task ('Self').

- :alternative(Self,[Constituent_task1,Constituent_task2,...])

  This predicate shows the relationship that 'Constituent_task1', 'Constituent_task2', ... are alternatives during the execution of task ('Self').

- :optional(Self,Constituent_task)

  This predicate shows that the execution of 'Constituent_task' is optional during the execution of task ('Self').

Constraints about attributes must be maintained during the execution of a task, for the execution of the constituent tasks may cause changes in the states. There are two kinds of constraints related to attributes. One is concerned with the values of attributes. However, this kind can be represented in the definitions about other kinds of objects (e.g., data objects). Therefore, it is excluded from the task descriptions. The other kind is concerned with the relationships between the attributes of a task and the attributes of the constituent tasks. To describe this relationship, the instance method predicate ":match(Self,Attribute0,[Constituent_task,Attribute1])" is used. This predicate shows the relationship that the value of 'Attribute0' of task ('Self') is equal to the value of 'Attribute1' of 'Constituent_task'. In Figure 2, the relationship defines the value of the attribute 'travel_expense_account_form' as equal to the value of the attribute 'form' of the constituent task 'fill_out_travel_expense_account_form'.

A primitive-task is defined in terms of the pre-condition, the post-condition, and the attributes as well as a composite-task. Also, the instance method predicate ":execute(Task)" is defined to perform its function.

4. HOW AN OFFICE TASK IS PERFORMED

This section discusses how an office task is performed based on the task descriptions explained in the previous section. Performance of an office task is regarded as the process in which a planning tree is built by recursively dividing a task into subtasks. Each node of the planning tree represents an instance of a task description and each arc between two nodes represents the relationship between a task and a subtask. This task-subtask relationship is regarded as the relationship between goal and subgoal in problem solving. A task represented by a leaf node is primitive. Its performance is implemented by the instance method predicate ":execute(Task)" that is defined in the task description. Figure 3 shows the top-most level of a meta-level procedure which schedules the performance of a composite-task based on its own task description. This procedure is implemented as an instance method predicate of 'composite_task' class, so that the subclasses which represent the composite-task are able to inherit it and so be capable of scheduling.

```
:execute(Self):-
    repeat,
    select_subtask(Self,Task,Parameter_list),
    execute(Task,Parameter_list),
    constraint_propagation(Self,Task),
    history_registration(Self,Task);
```

Figure 3. Top-most level of the scheduling procedure

The procedure in Figure 3 utilizes forward reasoning strategy using the task description to perform a composite-task. Side-effects caused by forward reasoning are reflected in the status of the related slot values of objects. When the procedure is invoked, the before-demon predicate of ":execute(Self)" is first called to check the pre-condition of this task. Cycles are then repeated until the post-condition of the task is satisfied. In each cycle, a subtask applicable to the current state is selected. Then, after its instantiation, it is recursively executed. Attribute values are propagated to maintain constraints after the execution of the subtask. Finally, data about the execution of the subtask is recorded in the history. When each cycle is finished, the after-demon predicate of ":execute(Self)" is called to check that the task is completed. If the post-condition is satisfied, the task is finished. Otherwise, the next cycle is started under the backtracking control of Prolog, without restoring any side-effects.

Selection of a subtask in each cycle uses the instance method predicate ":constituent_task", the constraints about the temporal ordering of the constituent tasks, and the history which holds the execution log. The office world is open-ended and evolving, so a situation may occur, in which the procedure cannot deal with the selection of a subtask using only task descriptions. When a subtask cannot be chosen, office workers must undertake to solve the problem, for they have more relevant knowledge than the computer. Our system enables office workers to intervene, whenever the system cannot select the next subtask.

In each cycle, values are propagated to maintain constraints using the relationships between the attributes of the main task and of its constituent tasks, for the status is changed by the execution of subtasks. After a subtask is completed in each cycle, the instance method predicate ":match(Self,Attribute0,[Constituent_task,Attribute1])" is called to ascertain if the propagation is necessary. If it is, the value of the 'Attribute1' of 'Constituent_task' is propagated to the 'Attribute0' slot of the task ('Self').

5. CONCLUSIONS

There are several efforts to develop a knowledge-based office system. Barber, et al. [1],[2] propose an excellent problem-solving method which deals with unstructured office tasks. POISE [3] provides office workers with a wide range of assistance. The task description framework of POISE is similar to ours. However, our work differs from POISE, because our system uses the logic-programming feature of ESP to enable natural description of the relationships. Woo, et al. [16] also make use of the object-oriented approach to modeling office work. They use production systems to cope with complex office tasks. To deal with such tasks, our system borrows the goal-oriented ability of Prolog as well as the help of the office workers.

This paper discusses intelligent support for office work. ESP is used to provide the object-oriented feature and the logic-programming feature. Both the features are sufficient to represent the knowledge of office work using a conceptually natural constructs. They are also appropriate to develop the meta-level procedures which refer to the object-level of the knowledge. Discussion is focused on modeling office tasks and on how office tasks are performed, for office tasks are the basis for making an office system intelligent and active. In our modeling framework, an office task is represented by a class of ESP. The predicate form of ESP is used to represent constituent tasks, pre-conditions, post-conditions, and constraints of the tasks. The meta-level procedures are implemented as method predicates defined in the system classes and inherited by classes representing office tasks for intelligent support.

ACKNOWLEDGEMENT

This work is based on the results of the R & D activities of the Fifth Generation Computer Systems Project of Japan. We would like to thank Mr. Y. Iwashita, the chief of the Fifth Section of ICOT (Institute for New Generation Computer Technology) for his encouragement and support. We also thank Mr. T. Hayashi, manager of Software Laboratory of Fujitsu Laboratories Limited, and Mr. Y. Hiratsuka, senior researcher of Fujitsu Laboratories Limited, for their continued guidance. Special mention should go to Dr. A. Makinouchi of Fujitsu Laboratories Limited for his helpful comments on the draft of this paper.

REFERENCES

[1]Barber, G., "Supporting organizational problem solving with a workstation", ACM TOOIS, Vol.1, No.1, pp.45-67, January, 1983

[2]Barber, G. et al., "Semantic support for work in organizations", Proceedings of IFIP 83, pp.561-566, 1983

[3]Croft, W. B. and Lefkowitz, L. S.,"Task support in an office system" , ACM TOOIS, Vol.2, No.3, pp.197-212, July, 1984

[4]Chikayama, T., "ESP referenc manual", ICOT TR-044, 1983

[5]Fikes, R. E. and Henderson, D. A., Jr., "On supporting the use of procedures in office work", Proceedings of 1st National Conference on Artificial Intelligence, pp.202-207, 1980

[6]Fikes, R. E., "Odyssey: a knowledge-based assistant", Artificial Intelligence, Vol.16, pp.331-361, 1981

[7]Genesereth, M. R. and Ginsberg, M. L., "Logic programming", Communication of ACM, Vol.28, No.9, pp.933-941, September, 1985

[8]Gibbs, S. and Tsichritzis, D., "A data modeling approach for office information systems", ACM TOOIS, Vol.1, No.4, pp.299-319, October, 1983

[9]Goldberg, A. and Robson, D., "Smalltalk-80: the language and its

implementation", Addison-Wesley, 1983

[10]Kowalski, R., "Logic programming", Proceedings of IFIP 83, pp.133-145, 1983

[11]Hogg, J. et al., "Form procedures", in Omega Alpha, Tsichritzis, D., ed., TR-CSRG-127, Computer Sys. Res. Group, Univ. of Toronto, pp.101-133, March, 1981

[12]Lochovsky, F. H., "Improving office productivity: a technology perspective", Proceedings of the IEEE, Vol.71, No.4, pp.512-518, 1983

[13]Maes, P., "Steps towards knowledge-based office systems", Proceedings of 1st conference on artificial intelligence applications, pp.562-568, 1984

[14]Shu, N. C. et al., "Specification of forms processing and business procedures for office automation", IEEE Trans. on SE, Vol.SE-8, No.5, pp.499-512, 1982

[15]Weinreb, D. and Moon, D., "Flavors: message passing in the Lisp machine", MIT AI Memo 602, November, 1980

[16]Woo, C. C. and Lochovsky, F. H., "An object-based approach to modeling office work", IEEE Database Engineering, Vol.8, No.4, pp.14-22, December, 1985

[17]Zloof, M. M., "QBE/OBE: a language for office and business automation", IEEE Computer, Vol.14, pp.13-22, 1981