PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT. © ICOT, 1984

AN OBJECT-ORIENTED APPROACH TO KNOWLEDGE SYSTEMS

Mario Tokoro and Yutaka Ishikawa

Department of Electrical Engineering
Keio University
3-14-1 Hiyoshi, Yokohama 223 JAPAN

ABSTRACT

A method for an object oriented modeling of knowledge systems called DKOM (Distributed Knowledge Object Modeling) is proposed. In this modeling method, a knowledge system consists of cooperative knowledge objects, where each knowledge object consists of a behavior part, a knowledge part, and a monitor part. An object oriented language called ORIENT84/K has been designed based on the DKOM. The behavior part of an object contains methods like those in Smalltalk; the knowledge part contains rules and facts like those in Prolog; and the monitor part monitors and controls the object. The relation between class and object, the relation between the behavior part and knowledge part, inference from knowledge, addition and deletion of knowledge, addition and deletion of methods, and access control of objects are described. An expert system is built using ORIENT84/K and the performance of ORIENT84/K is compared with some other programming languages/systems.

1. INTRODUCTION

Various knowledge-based systems have been built to understand intellectual processes in the fields of expert systems, robotics, natural language understanding, learning systems and so forth [Barr and Feigenbaum 1981]. In the development of most of these systems, first tailored software tools had to be produced to build individual objective systems [Hayes-Roth et al. 1983], and most of the systems were programmed in Lisp, which is considered to be a low level language for knowledge system implementation. Thus, demands for generalized programming languages/systems have arisen.

There have been several programming languages/systems proposed to support the building of knowledge systems. For example, KRL [Bobrow and Winograd 1976] and FRL [Goldstein and Roberts 1977] are based on **frames** and are implemented on Lisp. OPS-5 [Forgy 1981] is based on the **production system** and is implemented on Lisp. PIE [Goldstein and Bobrow 1981] is based on **frames** and is implemented on Smalltalk. LOOPS [Bobrow 1982] combined an **object-oriented paradigm** and a production-based **rule-oriented paradigm** on Lisp. LOOKS [Mizoguchi et al. 1984] and Mandala [Furukawa et al. 1984] have adopted **logic programming paradigm** approaches.

In order to build large knowledge systems, we need a programming language/system with (i) a wider application area, (ii) a higher descriptivity and maintainability, and (iii) a higher execution efficiency. With this in mind, we propose a new modeling method, called **Distributed Knowledge Object Modeling** (DKOM), for representing knowledge systems. In DKOM, a knowledge system is composed of distributed **Knowledge Objects** (KO), each of which consists of a **behavior part**, a **knowledge-base part**, and a **monitor part**. Knowledge objects run in parallel, and communicate with each other by message passing. A knowledge object makes decisions according to its knowledge in responding to a request message. It may send messages to other objects to ask their help in making a decision. It can acquire knowledge using inquiry messages and can generalize its knowledge. The monitor part monitors and controls all the activities of the object.

An object oriented language called ORIENT84/K has been designed based on the DKOM. This language provides the capability of describing the behavior of an object as the Smalltalk-80 [Goldberg and Robson 1983] system. It provides the capability of describing rules and facts as Prolog. A prototype of ORIENT84/K has been implemented on Franz Lisp. The final version will be implemented on an object oriented architecture which is being built [Ishikawa and Tokoro 1984]. It will provide capabilities for the execution of a knowledge system in a distributed multiprocessor environment.

In the next section, we discuss issues in generalized programming languages/systems for building knowledge systems and propose the Distributed Knowledge Object Modeling (DKOM). In section three, we describe an object oriented language ORIENT84/K with examples. In section four, we describe a knowledge system programmed in ORIENT84/K and the descriptive capability of ORIENT84/K is discussed in comparison with other programming languages/systems. In the last section, we conclude with remarks on the DKOM and ORIENT84/K and future plans.

2. DISTRIBUTED KNOWLEDGE OBJECT MODEL

2.1. Principal Objective

Let us first consider the intellectual behavior of a human being. We have acceptors such as eyes and ears to receive data, memory to record information, and actuators such as hands and vocal chords to output data. The process of our behavior could be simplified and described as follows:

(1) On accepting data through acceptors, we interpret the data in order to recognize them as information.

(2) We then infer from knowledge in our memory to make a decision. We may initiate actions through actuators to obtain further knowledge for making decisions. Or we may hypothesize and prove, and we generalize rules in making decisions.

(3) By using the decision made in (2), we initiate final actions that are usually irreversible.

(4) We monitor the effect of the actions made in (3) as feed-back for future decision making. We also monitor the process of inference in (2) to improve our decision making process.

There have been various discussions about knowledge systems and their description languages/systems, but mainly from the viewpoint of process (2). Since process (2) is the process of simulating the real world in a knowledge system, modeling from the viewpoint of process (2) is appropriate for most knowledge systems, especially for expert systems. In other areas of knowledge systems such as robotics, knowledge systems include all the above processes.

Our principal objective is to devise a new modeling scheme and programming language/system based on the modeling scheme for describing large knowledge systems: the programming language/system can simulate all of the four processes listed above. In the following subsections, we discuss some important issues in achieving our principal objective.

2.2. Execution Mechanisms

Representing a program in either a **declarative manner** or a **procedural manner** is an old yet important issue [Winograd 1975]. Writing a program in a declarative manner is usually very easy for knowledge systems in a well-defined area. On the other hand, it usually gives us less efficiency than a procedural representation. In addition, when we would like to control the execution of programs, programs become very complicated.

Representing a program in a procedural manner is suitable for describing behavior of an object. A procedural language can easily manipulate arbitrary data structures. However, it is not always appropriate to describe rules and facts. Therefore, we would like to utilize both representations and their execution mechanisms in our programming language/system.

2.3. Modularization Mechanisms

In representing knowledge systems, there are two levels of modularization:

(1) the **rule/fact level modularization** in which modular programming is achieved at the granularity of a rule or a fact, and

(2) the **object level modularization** in which modular programming is achieved at the granularity of an object.

The rule/fact level modularization premises that a knowledge system consists of a collection of knowledge fragments. Production system-based languages such as OPS-5 and predicate logic-based languages such as Prolog are examples of this modularization. Since each knowledge fragment can be treated independently from others, it is easy to append knowledge to and delete knowledge from a knowledge system. On the other hand, it is difficult to find relations among rules and facts. In addition, for a large knowledge system, conditions for each rule or fact tends to be complex.

The object level modularization premises that knowledge relating to an object should be contained in the object and that a knowledge system consists of a collection of such objects. Such modularization is favorable from the viewpoint of execution efficiency. However, it is sometimes difficult to represent general rules or interrelations among objects.

We would like to utilize both of the modularization. That is to say, we would like to describe a knowledge system as a collection of objects, where knowledge in each object is represented at the level of a rule or a fact.

2.4. Predicate Logic Approach and Object Oriented Approach

Prolog can be considered to be a predicate logic-based declarative language (rather than procedural language) with the rule/fact level modularization. The Smalltalk-80 system can be considered as a procedural language (rather than declarative language) with the object level modularization.

Although these languages appear in quite different manners, there is a duality relation between them. In predicate logic, predicate "m is Q to n" is described as

Q(m, n).

Thus, predicate Q knows and holds all the pairs of

X and Y which make this predicate true. In object orientation, object m provides the following method of implementing an equivalent effect:

```
Q: x | |
  x isNil ifTrue:[ ↑ n ]
  ↑ x == n.
```

The unification function of Prolog can also be described in the object oriented manner by using broadcast messages. For example, a Prolog clause

C(X, Y) :- P(X, Z), Q(Z, Y).

for a given x for X can be described in an object-oriented manner as follows:

```
C: y | z answer |
  answer ← OrderedCollection new.
  z ← nil.
  z ← self P: z.
  (z isEmpty) whileFalse:[
     (((z removeLast) Q: y) isNil)
        ifFalse:[ answer addLast: y]
  ]
  ↑ answer.
```

In predicate logic, it is natural to represent relations among concepts (objects) and it is powerful for deriving new relations from a given relation among concepts. It is, however, impossible to represent history sensitive characteristics, or states, of concepts. Thus, we need lists of characters to be passed between predicates in Prolog. In this sense, Prolog is used as a list processing language whose syntax is predicate logic oriented.

In object orientation, the abstraction of concepts is easily achieved by using class definitions and the instantiation of the object. Hierarchical abstraction is also achieved by using the notion of class inheritance. Thus, object orientation is suitable for representing the characteristics or properties of objects, including the time-varying states of objects. In object orientation, however, it is really difficult to represent relations among objects as we saw in the above example.

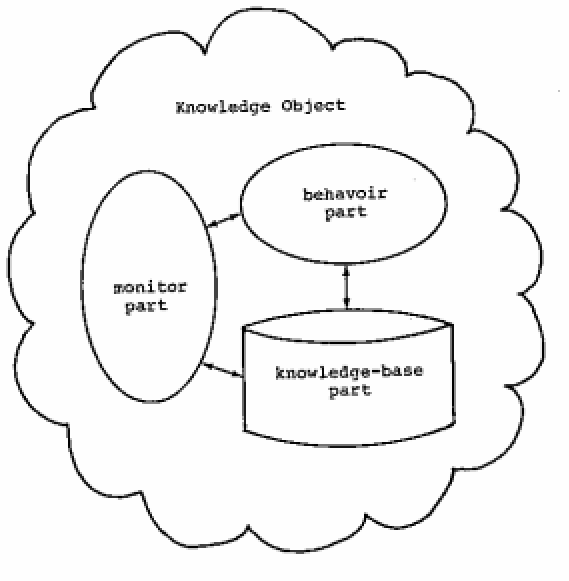


Fig. 1 The components of a Knowledge Object

2.5. Proposal of Distributed Knowledge Object Modeling

In concluding the above discussions, we propose a method of **Distributed Knowledge Object Modeling** (DKOM) for knowledge systems. In order to represent a large knowledge system in a simple and natural way, we consider that it should be composed of small knowledge systems, each of which can simulates all the processes described in subsection 2.1. Thus, in DKOM, a knowledge system consists of distributed **Knowledge Objects** (KO's). A knowledge object consists of **behavior part, knowledge-base part,** and **monitor part** (Fig. 1). It is created by its class. A class can have multiple super classes.

Knowledge objects run in parallel, and communicate with each other by message passing. A knowledge object makes decisions based upon its knowledge in responding to a request message. It may send messages to other objects to ask for their help in making decisions. It can acquire knowledge by inquiry messages and can generalize knowledge.

The behavior part is described in a procedural manner. It contains methods defined in its class and that inherited from its super classes. A method can be considered to be a procedure that describes an action of the object or as an attribute of the object. A method sends messages and manipulates its own variables in this object. There are some predefined methods (which are defined in class Object) for accessing the knowledge-base part. There are a few predefined methods for inferring from the knowledge in the knowledge-base part.

The knowledge-base part of an object is described in a declarative manner. It is the local knowledge-base of the object, containing rules and facts defined in its class, inherited from its super classes, and acquired through inquiry message to other objects. This part could be thought of as own variables of the object, except that there are predefined methods to infer from the knowledge in this part.

The monitor part is the demon for the object. It controls incoming messages, monitors the object's behavior and inferences, and improves the behavior and the knowledge-base of this object by using gathered statistics.

The following section describes an object-oriented language called ORIENT84/K, which is based on DKOM.

3. ORIENT84/K

In this section, an outline of the language is described with examples.

3.1. General Structure

The syntax and semantics of ORIENT84/K owe much to and are extended from Smalltalk-80. It has the **metaclass-class-instance** hierarchy, and the **multiple inheritance** from multiple super classes. All the objects run in parallel.

A class describes the common attributes and local knowledge-base of the instances of this class. Such an instance is called a knowledge object of the class. Knowledge objects communicate with each other by message passing.

The syntax and semantics of the knowledge-base part owe much to Prolog. We extended the kind of terms and modified the syntax in defining the interface with the behavior part. The capability of list processing is omitted, since list processing can be naturally described in the behavior part.

In the following subsection, the syntax and semantics of ORIENT84/K is described with using a modified BNF notation, where / is used for selection instead of usual | , { a } is optional, both {a}... and a... represent aaa...., and a{b}... represents aba...ba.

3.2. Class

A class definition consists of two sections: a **class section** and an **instance section**. The class section defines the class's **monitor part**, **behavior part**, and **knowledge-base part**. The instance section defines the instance's **monitor part**, **behavior part**, and **knowledge-base part**.

```
<class definition> ::=   CLASS <class name>
  INHERIT FROM <class name>...
  CLASS SECTION <section body>
  INSTANCE SECTION <section body>
<section body> ::=
    OWN VARIABLES <own variable definition>...
    MONITOR PART <monitor part>
    BEHAVIOR PART <behavior part>
    KNOWLEDGE-BASE PART
      <knowledge-base part>
<own variable definition> ::=
    <own variable>{ \ <class name> }
<own variable> ::= <variable name>
```

The declaration **INHERIT FROM** specifies the super classes of this class. Variables can optionally be **typed** by classes. The default type for a variable is **any**.

3.3. Monitor Part

The monitor part has the following functions: access control, prioritorized message handling, and statistics gathering. The specification of this part is not fixed, since this part largely relates to a system description language of the same ORIENT84/K language family.

```
<monitor part> ::=
  ACCESSIBLE FROM <access permission list>...
  PRIORITY <methods priority>...
<access permission list> ::= <object name>
    :<message pattern definition>...
<methods priority> ::= <priority level>
    :<message pattern definition>...
```

The declaration **ACCESSIBLE FROM** specifies to whom an instance shows which subset of methods of the instance. An access permission list can be added/deleted in the execution of the object by the **add_permission** <access permission list> and **delete_permission** <access permission list> predefined methods of class Object.

Arrival of a message with a higher priority method suspends the execution of any lower priority method which might be executing. The declaration of **active values** would be specified in the monitor part. A statistic gathering function should be provided in the monitor part so that the method part can utilize the statistical information for reorganizing methods and knowledge.

3.4. Behavior Part

The behavior part contains methods. Methods are the attributes of the instance which are seen by others. A method is executed by receiving a corresponding message, and may or may not return an answer to the caller. The execution of a message may change the state of the object.

```
<behavior part> ::= <method definition>...
<method definition> ::=
    <message pattern definition>
    |{ <temporary variable definition>... }|
    <statement>...
<temporary variable definition> ::=
    <temporary variable>{ \ <class name> }
<message pattern definition> ::= <unary selector> /
    <binary selector> <formal variable definition> /
    { <keyword> <formal variable definition> }...
<formal variable definition> ::=
    <formal variable>{ \ <class name>}
<temporary variable> ::= <variable name>
<formal variable> ::= <variable name>
```

In order to add and delete methods to and from the object, we have the following predefined methods: **add_method** <method definition> and **delete_method** <message pattern definition>.

3.5. Knowledge-Base Part

The knowledge-base part is the local knowledge base of the object. It contains the facts of the object, facts of other objects, and rules obtaining among objects which are defined in its class, inherited from super classes, and/or acquired by message passing. It can be considered to be the object's special own variables which contain rules and facts. Unlike in Smalltalk-80, the rules and facts defined in a class and acquired by message passing, together with inherited rules and facts, can be used by methods defined in the class and super classes. That is to say, the visibility of rules and facts through the super chains is equivalent to that of methods in Smalltalk-80.

```
<knowledge-base part> ::= <clause>...
<clause> ::= <fact definition> / <rule definition>
<fact definition> ::= <left hand formula> .
<rule definition> ::= <left hand formula>
        |{ <temporary variable definition>... }|
        <right hand formula>{,}... .
<left hand formula> ::=
    <predicate> ( <L-term>{,}... )
<right hand formula>::=<atomic formula> /
    self <message pattern>
<atomic formula> ::= <predicate>(<R-term>{,}...)
<L-term> ::= <string constant> / <own variable> /
    <K-formal variable definition>
<R-term> ::= <string constant> / <own variable>
    / <K-formal variable>
    / <temporary variable> / !
<string constant> ::= '<characters>'
<K-formal variable definition> ::=
    <K-formal variable>{ \ <class name> }
<K-formal variable> ::= ? <variable name>
```

Four kinds of terms are used in the knowledge-base part: a string constant, an own variable, a K-formal variable, and a temporary variable. As shown in Table 1, string constants correspond to constants in Prolog. K-formal variables corresponds to variables in Prolog and are used to pass information between the behavior part and the knowledge part. There are no correspondents of own variables in Prolog. Own variables are used to pass information from the behavior part to the knowledge part. When unified, an own variable acts as a constant in Prolog, since it has been bound to an object. In the clause of brother(?x, ?y), f declares a temporary variable as in the method part. Thus, temporary variables correspond to variables in Prolog.

A mechanism to call a method of an object from inside the knowledge-base part of the object is provided. By using this mechanism, preserving all the backtrack information, a rule can call a method for more information, and then continue inferences.

The contents of the knowledge-base part can be changed at any time by the execution of the predefined methods **addKB** (<clause>) **appendKB** (<clause>) and **deleteKB** (<clause>). The **addKB** and **appendKB** methods correspond to **asserta** and **assertz** of Prolog, respectively.

3.6. Interaction between Behavior and Knowledge-Base Part

As in Conniver [Sussman and McDermott 1972], a method can initiate inference by using the predefined methods **unify** and **foreach_unify**. Information is passed between the behavior part and the knowledge-base part through L-formal variables and own variables. Symbol ? is used to show that ?<variable name> returns a unified result from the knowledge-base part and this result is accessed by the <variable name> in the behavior part. The unify method returns one set of unified results for the L-formal variables. The foreach_unify returns all sets of unified results for the L-formal variables. It sends **value** for each set of unified result to the **block** that follows.

For example, a method which sends mail to all the brothers of somebody m is described in this model as follows:

```
INSTANCE VARIABLES
  john tom mike andy peter henry robert
BEHAVIOR PART
  sendToBrothersOf: m mail: mess1 | x |
    foreach_unify(brother(m, ?x))
      do:[ :x| → x sendMail: mess1].
KNOWLEDGE-BASE PART
  "RULE"
    brother(?x, ?y) | f |
        father(?x, f), father(?y, f).
  "FACT"
    father(john, henry).
    father(tom, robert).
    father(mike, henry).
    father(andy, robert).
    father(peter, robert).
```

The following is an example of changing the content of the knowledge base part. In order to acquire the facts of the mother-child relation, the following expression, which yields broadcasting, is executed:

```
askMother | m x |
  foreach_unify(name(?x))
    do:[:x | m ← x mother,
        appendKB(mother(x, m))].
```

Table 1 Correspondence between ORIENT84/K variables and Prolog variables

ORIENT84/K	Prolog
`father(?x, 'y').`	`father(X, y).`
`father(x, y).`	none
`brother(?x, ?y) | f | father(?x, f), father(?y, f).`	`brother(X, Y) :- father(X, F), father(Y, F).`

3.7. Synchronization

In Smalltalk, objects are not executed concurrently. Thus, a message is sent and the object blocks until the receiver returns the results. In contrast, objects are executed concurrently in DKOM. In order to maximaly utilize the capability of this modeling, we added a syntax for the non-blocking message send that does not need the result to be returned. Thus, <expression> can be in the form:

→ <object name> <message pattern>.

The non-blocking message send is effectively used to send a broadcast message as shown in one of the previous examples.

4. BUILDING AN EXPERT SYSTEM

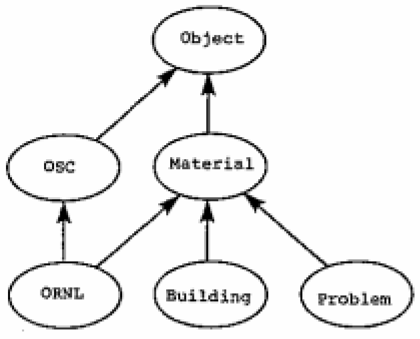
4.1. Program

In order to examine the descriptive capability of ORIENT84/K, a well-known expert system problem, the building of an expert system for the emergency management of inland oil and hazardous chemical spills at the Oak Ridge National Laboratory ORNL, was chosen. This expert system was described in EMYCIN, KAS, EXPERT, OPS-5, ROSIE, RLL, and others, in order to compare their descriptive capability [Hayes-Roth et al. 1983].

This problem is classified as a crisis-managent problem. When a discovery of spills of oil, hazardous chemical, or base is reported, the expert system locates the source of the spill, identifies the spilled material, estimates the quantity of the spill flow, evaluates the hazards, notifies inhabitants, designates countermeasures for the spill flow, and reports to responsible authorities. The expert system possesses knowledge of geographical information, quantities and kinds of materials in storages, characteristics of the materials, countermeasures, regulations concerning the hazards of chemical materials, and so forth. In order to pursue these tasks, the expert system infers from this knowledge, asks the reporter, and frees men for getting more information.

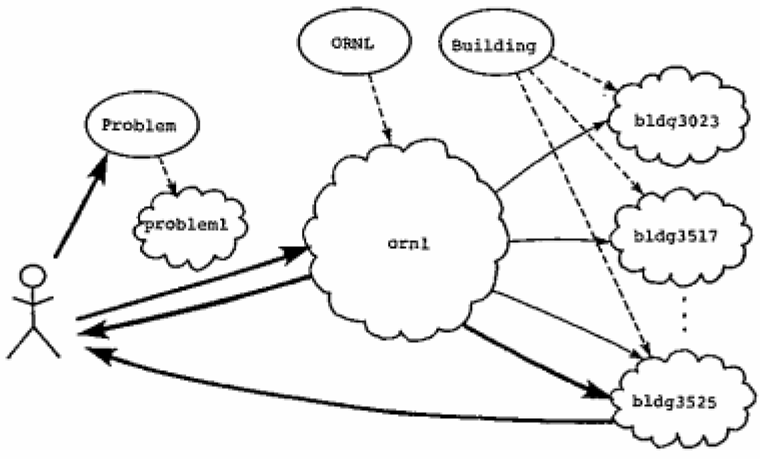
We built a simplified version of this expert system with using ORIENT84/K. The program for this expert system consists of five classes: class Problem defines an object which receives the discovery of spills, class ORNL defines an object which advises about the way to find the source of a spill and to identify the material, class Building defines the storage of materials and countermeasures, class OSC defines the inference method for finding the source, and class Material defines the characteristics of materials. The hierarchical relation among these classes is shown in Fig. 2.

In execution, a user sends messages to class Problem to create an instance problem1, which contains the information about the discovery of a spill. Then, the user sends instance problem1 to the instance ornl of the class ORNL. The instance ornl analyzes the information and advises the user for locating the source of the spill. According to the advices, the user gets more information, and reports back to the instance ornl. When the sources are determined, the instance ornl sends messages to an instance bldgXXXX, which is an instance of class Building to perform countermeasures. Then, the instance ornl sends the user advice for locating other sources of spills. Fig. 3 shows the relation of objects in execution. Fig. 4 shows a part of the class definitions.



An arrow indicates the superclass-subclass relation

Fig. 2 The hierarchical relation among the class



An arrow indicates that the object holds an object. A boldface arrow indicates that the object sends a message to an instance or a class. A dotted arrow indicates that the object is created by the class.

Fig. 3 The relation among objects in execution

```
CLASS                  OSC
   INHERIT FROM        Object
   CLASS SECTION
                       .
                       .
   INSTANCE SECTION
                       .
                       .
      KNOWLEDGE-BASE PART
                       .
                       .
        findSource(?bldg, ?source, ?smat) | loc tbldg |
                flow_loc(loc), find(?source, loc),
                       source(tbldg, ?source),
                       storage(?smat, ?source), obj(?bldg, tbldg).

        find(?source, ?dest) | yy |
                arc(yy, ?dest), find1(?source, yy).
        find(?source, ?dest) | mat |
                posmat(mat), storage(mat, ?source),
                       !, self checkpoint: ?dest.

        find1(?source, ?dest) | tt |
                merge(?dest), !, arc(tt, ?dest), fss(tt),
                       !, self checkpoint: ?dest, find(?source, ?dest).
        find1(?source, ?dest) | |
                fss(?dest), find(?source, ?dest).
                       .
                       .

CLASS                  ORNL
   INHERIT FROM        OSC Material
   CLASS SECTION
                       .
                       .
   INSTANCE SECTION
      BEHAVIOR PART
                       .
                       .
        call: problem | bldg loc mat |
                       .
                       .
                foreach_unify(findSource(?bldg, ?loc, ?mat)) do: [
                 :bldg |   ->bldg countermeasure: loc.
                ]
                problem set_material: mat.
                       .
                       .
        checkpoint: point [ |
                Terminal print: 'Please inspect '.
                Terminal print: point.
                Terminal print: ' Is the liquid flowing in '.
                Terminal print: point.
                Terminal print: '? '.

                (Terminal getstring) = 'y'
                       ifTrue: [
                               appendKB(flow(point)).
                               ↑true.
                       ]
                       ifFalse: [ ↑false ].
                       .
                       .
```

Fig. 4 The description of an expert system in ORIENT84/K

4.2. Discussion

As we see the part of the program of this expert system in Fig. 4, ORIENT84/K naturally describes the problem in both the object-oriented paradigm with message passing and the predicate logic-based paradigm.

The same expert system has been programmed using Prolog. The structure of the part of the program for inference was the same as that in the class ORNL. Since Prolog does not support the object-oriented paradigm, we used a file for each instance in the ORIENT84/K program and reconsulted the files as the program proceeded. One of the weak points in Prolog is the difficulty of naming predicates uniquely. In the other words, there is no visibility control in Prolog. Thus, even though we store clauses in different files, all of the clauses are flat at the time of execution. This caused program bugs. On contrary, ORIENT84/K provides a knowledge-base for each object. Therefore, it was easier to write this expert system in ORIENT84/K. ORIENT84/K also surpassed Prolog in writing parts other than unification.

We also described the same expert system in LISP. In the LISP program, we stored the geography of the drain flows in a list and searched the possible sources of the spill on the list. It seems to be difficult to change the list when new buildings, new drains, or new materials were added.

When we would implement the same expert system in Smalltalk, we should define a class which provides methods of inference and the structure to retain knowledge. The program would look similar to that written in ORIENT84/K. The essential difference, however, is that the inference mechanism and the knowledge-base are defined together in a class in Smalltalk, while they are contained in an object or distributed in multiple objects in ORIENT84/K.

5. Conclusion

In this paper we proposed the method for the Distributed Knowledge Object Modeling (DKOM) and described the outline of an object oriented language called ORIENT84/K as a realization of this modeling method.

The modeling method views a knowledge system as the composition of cooperative knowledge objects, where each object holds its knowledge-base and communicates with other objects by message passing in order to acquire knowledge. These features coincide with the human activities of communicating, acquiring knowledge, making decisions, and doing actions.

The language provides the rule/fact level modularization and the object level modularization. Thus, we can build knowledge systems in the object-oriented manner and the predicate logic-based rule-oriented manner. Each knowledge object consists of the monitor part, the behavior part, and the knowledge-base part. The behavior part can be considered as a meta function to the knowledge-base part, and the monitor part that interfaces other objects (or worlds) as a meta function to the behavior part. With a little experience in writing programs in ORIENT84/K, we believe such a combination facilitates building large knowledge systems in various fields.

The prototype of the ORIENT84/K system is implemented in Franz Lisp and running on the Unix† operating system. The specification of the language is not complete, especially for the monitor part. This is partially because we would still like to improve the language through the feedback of writing programs. The other reason is because ORIENT84/K is a member of the ORIENT84 family of languages which run on a distributed object-oriented architecture [Ishikawa and Tokoro 1984], and we need to keep coherence throughout the family languages.

The distributed object oriented architecture has been designed and its software simulator is running also on the Unix operating system. Implementation of the ORIENT84/K system on this architecture is about to start, besides the design of some other family languages and the modification of the architecture to execute ORIENT84/K programs efficiently before siliconization planned in 1986.

ACKNOWLEDGEMENT

The authors are indebted to Motoo Kawamura and Takeo Maruichi for their help in the implementation of ORIENT84/K as well as for their valuable comments. The authors are also grateful to the members of ICOT WG2; the discussion with the members on wide spectrum of topics in knowledge representation, languages, and systems inspired this work.

REFERENCES

Barr, A. and Feigenbaum, E.A., "The Handbook of Artificial Intelligence", Volume 1, 2, and 3, William Kaufmann, Inc., 1981, 1982, 1982.

Bobrow, D.G. and Winograd, T., "An Overview of KRL, a Knowledge Representation Language," CSL-76-4, Xerox PARC, July 1976.

Bobrow, D.G., "The LOOPS Manual," Palo Alto Research Center Xerox PARC, KB-VLSI-81-13, 1982

Forgy, C., "The OPS-5 User's Manual," Technical Rept. CMU-CS-81-135, Computer Science Dept., Carnegie-Mellon Univ., 1981.

Furukawa, K. Takeuchi, Yasukawa, H, and Kunifuji, S., "Mandala : A Logic Based Knowledge Programming System," FGCS '84, ICOT, 1984.

Goldberg, I. and Robson, D, "Smalltalk-80: The Language and its Implementation," Addison-Wesley Publishing Co., 1983.

† Unix is a trademark of the Bell Laboratories.

Goldstein, I.P. and Roberts, R.B., "NUDGE: A Knowledge-Based Scheduling Program", IJCAI 5, 1977.

Goldstein, I.P. and Bobrow, D., "An Experimental Description-Based Programming Environment: Four Papers," CSL-81-3, Xerox Palo Alto Research Center, 1981.

Hayes-Roth, F., Waterman, D.A., and Lenat, D.B., "Building Expert Systems," Addison-Wesley Publishing Co., 1983.

Ishikawa, Y. and Tokoro, M., "The design of an object oriented architecture," Proc. of the 11th Int'l Symp. on Computer Architecture, Jun 1984.

Mizoguchi, F. Katayama, Y., and Owada, H., "LOOKS: Knowledge Representation System for Designing Expert System in the Framework of Logic Programming," FGCS '84, ICOT, 1984.

Sussmann, G.J. and McDermott, D.V., "From PLANNER to CONNIVER: A genetic approach," AFIPS, 1972.

Winograd, T., "Frame representations and the declarative/procedural controversy," in Representation and Understanding studies in Cognitive Science, Bobrow, D.G. and Colins, A., eds., Academic Press, 1975.