ICOT: AN INTEGRATED C-OBJECT TOOL FOR KNOWLEDGE-BASED PROGRAMMING

HYUNG JEONG YANG*, YEONGHO KIM** and JAE DONG YANG*

** Department of Computer Science Chonbuk National University*

Chonju, 560-756, South Korea E-mail: jdyang@jbdblab.chonbuk.ac.kr

*** Department of Industrial Engineering Seoul National University*

Seoul, 151-742, South Korea E-mail: ykim@cybernet.snu.ac.kr

In this paper, an Integrated C-Object Tool, namely ICOT, is proposed for knowledge-based programming. A major drawback of current rule-based expert system languages is that they have difficulty in handling composite objects as a unit of inference. An object-oriented model is a powerful alternative to complement the drawback. Each of these alone cannot capture all the semantics of knowledge, particularly in complex engineering domains. For a knowledge-based approach to be effective, both the object-oriented paradigm and the rule-based mechanism may need to be integrated into one framework. The framework may also need to support manipulation of fuzzy knowledge to model the real world as close as possible. Three types of fuzzy information are identified, and a proper way of representing and inferencing them is developed. ICOT provides a new framework into which rule-based deduction, object-oriented modeling, and fuzzy inferencing are combined altogether. This can become especially useful for developing knowledge-based engineering applications.

Keywords: Object-oriented model, Object inference, Knowledge-based programming, Fuzzy logic

1. Introduction

For the last three decades a number of rule-based expert system languages have been developed, and many of them have been successfully used in a broad range of application problems including diagnosis, signal understanding, troubleshooting, classification, and so on. These successes have rapidly widened their application area to more complex engineering support systems, such as GIS (Geographic Information System), CAD (Computer Aided Design), CAM (Computer Aided Manufacturing), and CASE (Computer Aided Software Engineering). These engineering applications frequently deal with composite objects, and often involve multimedia data, such as images, graphics, voice, and video and spatial data types, such as points, lines, polygons, etc. Additionally, the information or the knowledge processed in such applications is very often uncertain.

It, therefore, is a basic requirement that the languages for engineering applications be capable of representing and processing composite objects, various data types, and uncertain knowledge. However, many currently available rule-based expert system

languages have difficulties in handling composite objects as a unit of inference and limitations in supporting uncertainty and such a variety of data types. This has been pointed out as a major research issue by a number of research workers including [1, 2, 3].

Proposed in this paper is a composite object inference model that integrates an object-oriented paradigm and a traditional deductive mechanism. Although a deductive model has been predominantly used in many expert systems, the model alone cannot keep track of all the semantics of knowledge contained in complex engineering applications. It is generally recognized that an object-oriented model is able to effectively represent composite objects. It thus can be a promising approach for a deductive mechanism to have the modeling power of the objected-oriented design [4, 5]. Such an approach may facilitate the development of knowledge-based systems for engineering applications.

While modeling and processing various types of information, many systems assume that the information be completely and precisely known. However, the information dealt with in the real engineering world inherently contains uncertainty, and further is not always complete[6]. Simply ignoring the uncertainty, the systems may suffer from lack of rational results. To represent the vagueness of the real world more naturally and to derive more useful results, it can be essential to have a proper way of representing and processing uncertain information. A great number of researches, including [7, 8, 9] that incorporate uncertain information into database systems and expert systems, have been carried out. A major purpose of these previous researches is to devise data types strengthening semantic expressiveness of the uncertain information and to develop efficient methods of processing the data. It is widely conceived that Zadeh's fuzzy logic [10] provides a good tool for these purposes. Several expert system shells have been developed using the fuzzy logic to provide a quite natural interpretation of the uncertain information [11, 12].

In this research, a knowledge-based programming tool, namely ICOT (Integrated C-Object Tool), is developed. ICOT uses a new framework where an object-oriented paradigm and fuzzy logic are combined with a rule-based language. The object-oriented paradigm can enhance the modeling power of traditional rule-based systems, and the fuzzy logic can provide a proper representation and process of uncertain knowledge. An additional characteristic of ICOT is that the combination is made in such a way that it is compatible with an extant relational database. ICOT objects can be persistently stored in the database while exploiting most database facilities. The features of object-oriented paradigm are expressed in terms of relational constructs. The fuzzy reasoning process is also tailored and very well suited to the framework of the database. The proposed approach has been implemented, and successfully applied to a GIS application problem.

The format of this paper is as follows. First several previous research results are reviewed in Section 2. Then, a GIS route management problem, which all the illustrative examples used in this paper are drawn from, is briefly described in Section 3. The basic design and implementation of ICOT is addressed in Section 4. Methods of declaring classes and instances are respectively described in Section 4.1 and 4.2, and declaration of rules is in Section 4.3. Finally, the basic design is extended to fuzzy inference in Section 5, with conclusions in Section 6.

2. Related Work

Considered in this paper is an integration of object-oriented paradigm and fuzzy logic into a rule-based language. This approach has gained a great amount of attention from many research workers, and thus a huge body of literature related with this issue is available [3, 11, 13, 14]. In this section, some of the important previous researches that are widely used for applications or have a particularly interesting variation of an issue under discussion are reviewed in three folds. First, rule-based mechanisms adopting object-oriented paradigm are described. Second, expert systems using fuzzy logic are presented. Finally, C-ECLPS (C - Enhanced Common LISP Production System), the underlying framework of ICOT implementation, is introduced.

2.1 Integration of object-oriented paradigm and rule-based mechanism

Objects become a uniform programming element for general computing. As object-oriented programming has taken hold in the mainstream of knowledge engineering tools, a considerable amount of interest has been expressed in combining rule-based systems with object-oriented models. On the one hand, great emphasis has been placed on the implantation of object-oriented concepts in logic languages [15, 16, 17]. A method is represented by a logic language program within an object, and a message is represented by a goal that should be satisfied. This easily enables an object to have inference capability. However, the use of logic languages to implement inheritance control and message processing which are the basic object-oriented concepts cannot take advantage of the declarative properties of logic [18].

On the other hand, a number of rule-based database programming languages have been expanded by adopting object-oriented concepts. Illustra, a commercial product of POSTGRES, is an object-oriented database [19]. This system supports abstract data types which can be used to represent a rule as an attribute value. Dayal [20] represented a rule as an object having its own attributes and methods. Since these two approaches supports no general deductive rules and rely on procedural properties, their inference mechanisms require a significant amount of users' control. Maier [21] proposed a formal framework for object inference in a rule-based paradigm. This object inference is limited in that no provision is made for supporting composite objects and methods. Coral++ is expanded from Coral which is a rule-based query language [3]. While the language is capable of preserving the declarative properties of logic, it doesn't support multimedia data types and spatial data types. CLIPS 6.0 is an object-oriented version of CLIPS, recently developed at NASA [22]. CLIPS 6.0 supports modularized construction of knowledge base and pattern matching between the left hand side of rules and instances of user defined classes. However, the semantics of this language is significantly different from the behavioral semantics of its underlying database system. This semantic difference, which is usually referred to as an impedance mismatch problem, is its major limitation. K is another database programming language intended to overcome the impedance mismatch problem [13]. K serves as an interface to a high-level knowledge model of OSAM* [23]. A knowledge base can be defined, queried, and manipulated, and knowledge-based application systems can be coded. However, K does not provide any means for processing uncertain information, which is discussed in the next section.

2.2 Fuzzy logic in expert systems

Processing uncertain information in expert systems have also been an important research direct, and thus a number of approaches have been proposed. It is well known that the fuzzy logic proposed by Zadeh [10] is a good alternative to handle uncertain knowledge in rule-based expert systems [11, 12, 14]. In general, a fuzzy inference engine tries to match uncertain knowledge with the condition part of rules. When a rule is matched to some extent, if not exact, a conclusion is derived considering the degree of match.

Several fuzzy expert system shells have been developed to integrate the fuzzy logic into their reasoning processes. For example, FLOPS and Z-II are fuzzy expert systems. FLOPS [11], an extension of OPS5 [24] by adding fuzzy concepts, selectively executes rules according to their confidence factors. Z-II uses VAX-Lisp, and inferences based on the compositional rule of inference [12]. Recently, the implementation of Fuzzy CLIPS has been reported from NASA [25]. This enables domain experts to express rules using their own fuzzy terms. It also allows any mix of fuzzy and normal terms, numeric-comparison logic controls, and uncertainties in rules and facts.

These fuzzy expert systems, however, provide no object inference facility and inheritance mechanism. The object inference facility is a key to enhancing modeling power, and the inheritance mechanism is the essence to take advantage of code reusability. These two limitations may prevent the systems from being successfully employed in complex engineering applications. Furthermore, the semantics and syntax of these expert system shells differ from those of extant databases. Such incompatibility may make it difficult for their fuzzy reasoning ability to be integrated into the databases.

2.3 C-ECLPS (C - Enhanced Common LISP Production System)

ICOT is implemented on C-ECLPS [26], which is a C version of ECLPS [2]. ECLPS, a significant extension of OPS5, is a production system language developed to build rule-based expert systems using IBM Common LISP. The structure of OPS5 is basically similar to a relational database framework. An overview of the basic features of C-ECLPS are presented.

The inference engine uses facts and rules. A fact in the working memory is defined by a composition of a class name and attributes. An attribute is specified by its name and a value. The syntax specification of C-ECLPS is shown below. The first is to declare a class of facts, and the second to create a fact or an instance of a class.

```
(defwme  class_name  attribute_name  .....)
(make  class_name  attribute_name  attribute_value  .....)
```

A rule consists of patterns (LHS: Left Hand Side) and actions (RHS: Right Hand Side). There is no limit to the number of patterns and actions. The place where rules are stored is called a rule memory. The following defrule syntax is used to store a rule in the rule memory.

```
(defrule
(pattern1  .....)
(pattern2  .....)
(.....)
```

=>
(action1 .....)
(action2 .....)
(.....))

The process of C-ECLPS inference is shown in Figure 1. The inference utilizes forward chaining which first tries to match LHS parts of rules with facts in working memory, and then executes the RHS parts of matched rules. This is called an R-A (Recognize-Act) cycle. The Rete match algorithm [27] is used to match facts with a specific pattern. The R-A cycle comes into play only when the working memory is changed.

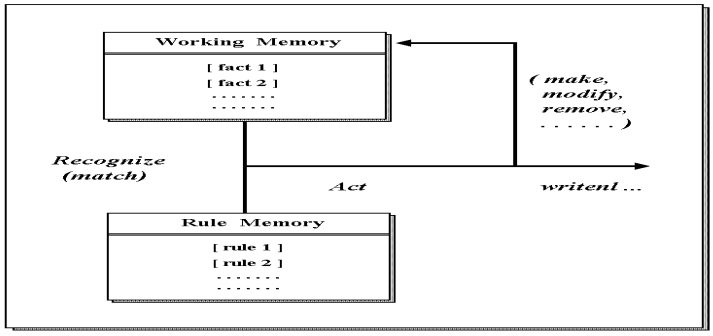

Figure 1. C-ECLPS Inference Process

A major advantage of C-ECLPS is that existing C library functions can be used in it and that knowledge can be input and inferenced in the environment of C programming language. That is, a user's application program and a C-ECLPS program are compiled and linked together. The same sort of feature is found in OPS5. A C program can also use C-ECLPS's rules of inference. The program can call the rules as it does subroutines. This interactivity between a C program and a C-ECLPS program is similar to that between CLIPS and LASER [28].

3. A GIS Example

An application of GIS route management problem is briefly described. The examples used in the following sections to explain the design and implementation of ICOT are taken from the application. Buses and subways are a major means of public transportation in metropolitan areas, such as New York City, London, Seoul, etc. Determining a traffic route network which can ensure an efficient operation of the bus-subway systems is critical to solving traffic congestion problems. The ICOT system has been developed to support the bus-subway route management.

The GIS route management system needs to deal with many types of geographic information that usually involves spatial data as well as non-spatial ones. The various types of data are modeled using an objected-oriented modeling technique. Shown in Figure 2 is a part of the schema of the geographic information which needs to be considered. A class is represented by a rectangle where its class name and attributes are shown. An arc represents hierarchical relationship between two classes. Two hierarchical structures are found in the

schema. One is a generalization hierarchy, linked by a thick arc. It represents a general-to-specific relationship between a class and its subclasses. For example, 'commuter_bus_route' and 'subway_route' are the subclasses of 'route'. The other is a composition hierarchy which is linked by a thin arc. This hierarchy interrelates an attribute and a class. For example, the 'source_node' attribute in 'route' and the 'station' class are of a composition hierarchy.

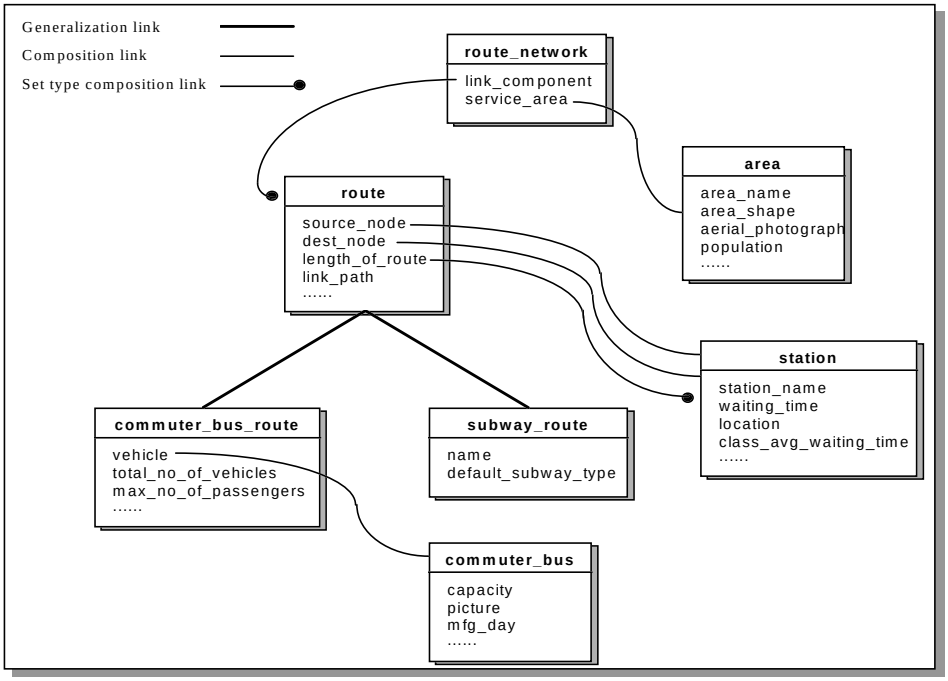

Figure 2. An Example Schema of Route Management

The inference in ICOT is performed using rules. The domain knowledge that is used to determine the configuration and management of traffic route networks has been acquired from experts working in the field. A large number of rules that are used in practice are collected. These include the route utilization rate over time, the maximum, minimum, and average speed of vehicles, the interconnection or transfer conditions, and so on. Due to space limitation, only a few of the rules are shown as examples in the rest of this paper. Once the object-oriented data models and the rules have been set up, it can be declared in the ICOT system.

4. Basic Design and Implementation of ICOT

ICOT is an extension of C-ECLPS by furnishing it with the features of object-oriented models and the functions of fuzzy inference. Discussed in this section is a method of combining object-oriented concepts and C-ECLPS. A mechanism to employ fuzzy logic in this combination is presented in the subsequent section.

There are in general two possible approaches to mounting new concepts or functions on an existing system. One is to translate new codes into equivalent codes that can be understood by the underlying system [15, 18], and the other to rebuild the kernel of the underlying system [16, 17]. The ICOT implementation has mainly adopted the first approach because its constructs are designed to have few semantic gaps with C-ECLPS. A preprocessor has been developed to translate ICOT codes into C-ECLPS. A modified Rete match algorithm is added on top of the C-ECLPS kernel. The algorithm is capable of dealing with several concepts, such as object equality, abstract data types, and fuzzy inference. These are not provided in C-ECLPS. ICOT has been implemented in C and runs on Sun Workstation. Three main issues in designing ICOT are provided below.

First, it is required to maintain the semantics implied by class hierarchies and composition hierarchies when translating ICOT declarations into C-ECLPS. For example, when an attribute value inherited from a superclass needs to be changed, the superclass should be identified. The translation creates an *isa* catalog and a composition catalog to carry super/subclass relationships and composite attribute/domain class relationships, respectively. Second, the Rete match algorithm is extended to perform object-based inference. Since the inference takes into account class hierarchy, it refers the information provided by the *isa* and composition catalogs mentioned above. Third, the fuzzy inference capability of ICOT has been implemented by allowing for the Rete match algorithm to perform inexact matching. Evaluation methods associated with fuzzy operators and inference units are provided. Possibility attributes in patterns are first added to the translated C-ECLPS codes, and then rules are evaluated by the modified algorithm.

C-ECLPS is basically a rule-based language. In order to extend it into an object-oriented model, several essential concepts of object orientation are required. First, to support generalization hierarchy structures, *isa* relations need to be specified in the declaration of classes. Secondly, to support composition hierarchy, a type mechanism is used in attribute domains. This allows for a class type to be assigned as an attribute type. That is, an instance object identifier of a class can appear as an attribute value in another class. Thirdly, when declaring a rule, a variable can be utilized for a class name. This provides the capability of navigating composition hierarchies. Once a variable is assigned to a class name in a pattern, the variable binds to an object identifier of an instance which belongs to the class. Such binding makes it easy to implement the composition hierarchy link. In other general cases, a rule can be applied to every instance of corresponding classes and of their subclasses. Taking a rule for a method enhances reusability of codes in a generalization hierarchy structure. These concepts are further detailed with the syntax specifications of ICOT.

4.1 Declaration of extended classes

The next specification syntax declares extended classes to support generalized composition hierarchy structures.

(defclass <class_name> isa <class_name> {<attr_def>}+)
<attr_def> = <special_attr> | <normal_attr>
< special_attr> = ∗<special_attr_desc> '_' <attr_name> ':' <attr_type>
<special_attr_desc> = common | class | default

<normal_attr> = ∗<attr_name> ‘:’ <attr_type>

In the syntax, an attribute can have various types, such as set, class, multimedia, and spatial data, as well as basic types, such as integer, real, string, etc.

<attr_type> =
<basic_type>|<set_type>|<disjunctive_type>|<tuple_type>|<class_type>|<multimedia_type>|
<spatial_type>|<method_type>

The meaning of each type is as follows.

- Basic type: This includes integer, real, character, and string.
  <basic_type> = int | real | char
- Set type: A set type allows repeating values.
  <set_type> = ‘{‘ <attr_type> ‘}’
- Disjunctive type: An attribute type is declared by a set of multiple types.
  <disjunctive_type>=‘{‘ <attr_type> ‘|’ <attr_type> ‘|’ ....‘}‘+
- Tuple type: A tuple consists of several composite sub-attributes.
  <tuple_type> = ‘[‘ {<attr_def>}+ ’]’
- Class type: A class can be used as an attribute type.
  <class_type> = <class_name>
- Multimedia type: ICOT supports multimedia data, including image, audio, and video data.
  <multimedia_type> = image | audio | video
- Spatial data type: ICOT supports spatial data, such as lines, points, and polygons.
  <spatial_type> = line | point | polygon
- Method type: A method is a set of executable codes. The method has arguments taking <attr_type> as their types and a result type which can be one of void, <basic_type>, and <class_type>. The result type is optionally specified.
  <method_type> = ‘(‘ <result_type> ‘)’ <method_name> ‘(‘ <method_argument> ‘)’

All the example classes shown in Figure 2 are defined below. After the 'defclass' keyword, a class name, 'isa' keyword, a superclass name, and attributes appear in this order.

(defclass route_network isa object ∗link_component:{route} ∗service_area:area)
(defclass route isa object ∗source_node:station ∗dest_node:station ∗length_of_route:int
∗link_path:{station} ∗common_supervisory_office:char_dept_of_transportation
∗avg_speed:int ∗no_of_ passengers:int ∗overtraffic:int ∗traffic_portion:real
∗degree_of_congestion : real ∗dispatch_interval : int ∗max_no_of_passengers:int)
(defclass station isa object ∗station_name:char ∗waiting_time:int ∗location:point ∗class_avg_
waiting_time:int ∗avg_waiting_ time:compute_avg_waiting_time(waiting_time))
(defclass area isa object ∗area_name:char ∗population:int ∗aerial_photograph:image
∗area_shape:polygon ∗write:write() ∗dispaly_photograph:display_aerial_photograph(image))
(defclass commuter_bus_route isa route ∗vehicle:commuter_bus ∗total_no_of_vehicles:int
∗max_no_of passengers:int ∗default_bus_type:char_city_bus
∗temp_vehicle:(commuter_bus)avail_vehicle(length_of_route))
(defclass subway_route isa route ∗name:char ∗default_subway_type:char_subway)
(defclass commuter_bus isa object ∗capacity:int ∗picture:image ∗mfg_date:[∗year:int ∗mon:int
∗date:int] ∗default_bus_type:char_city_bus)

In the definition of the 'route' class, the domain of 'source_node' and 'dest_node' is 'station' which is a class type. The type of 'link_path' is a set of station classes. A tuple type

is used for 'mfg_date' in the 'commuter_bus' class. The 'aerial_photograph' type contained in 'area' is image, a multimedia type. Many object-oriented systems treat a multimedia data type as a primitive class having an image type [1, 29], an audio type, and a video type as its subclasses. In ICOT, to preserve the inferencing power of C-ECLPS without radically modifying its framework, the multimedia data type is treated as an abstract data type rather than a class. The 'display_aerial_photograph' attribute in 'area' is a method type. The method type is further detailed later in this section.

It is sometimes needed to declare attribute values for an entire class rather than each individual instance. A property that is commonly applied to every instance of a class is called a *common attribute*. For example, if a supervisory office manages every instance of a route class, the office can be a common attribute of the class. This attribute is shown in the 'route' class definition. A *default attribute* is to represent a default value. A common attribute disallows for an individual instance to possess an exceptional value, whereas a default attribute does allow by explicitly specifying it. The attribute 'bus_type' in the 'commuter_bus' class is declared as a default attribute, so that a default value is applied to an instance when the attribute remains unspecified. Finally, an aggregate property, which can be computed from all the instances of a class, is a *class attribute*. 'Avg_waiting_time' in the 'station' class is a typical example of such an attribute. The value of 'avg_waiting_time' is derived from the 'waiting_time' values of all the instances of the 'station' class. To specify the common, default, and class attributes, 'class', 'common', and 'default' should precede the attribute names, respectively.

Several method type attributes are found in the examples. There are two ways of defining a method. One is introducing a new construct [15, 30], and the other regarding a method as an attribute [3]. The latter is used in this paper since executable codes can be an attribute value in C-ECLPS.

Methods can be classified into class method and instance method. A class method is concerned with an aggregate property of an entire class, while an instance method deals with some properties that are pertained to individual instances. For example, 'compute_avg_waiting_time' in the 'station' class is a class method. When this method is invoked, it carries out computing an average waiting time based on the 'waiting_time' attribute values of all instances in the 'station' class. This behavioral semantics is similar to that of SQL aggregate functions. The 'avail_vehicle' method shown in the 'temp_vehicle' attribute of the 'commuter_bus_route' class is an instance method. This method selects a temporally available commuter bus based on the length of one specific commuter bus route. Notice that the result type is 'commuter_bus' class. The method returns the object identifier of an instance in 'commuter_bus'. Both of these two types of methods are retained in class definitions. A method, once defined in a class, can be shared by all of its instance irrespective of the method type.

4.2 Declaration of instances supporting composite objects

An instance declaration consists of a class name and attributes, as in C-ECLPS. The class name indicates the class which the instance belongs to. An attribute is specified by its name and value. The value should follow the type specified in the class definition. A composite object is a group of interconnected objects that are instantiated together. If an instance *a*

requires another instance *b* for its attribute value, *a* is called a composite object. The next syntax is to declare such a composite object.

[<var>](make <class_name> {∗<attr_name> ':' <attr_val> | (make <class_name> {∗<attr_name>':' <attr_val> }+) }+)

Provided below are some examples of instance declaration. Each of these strictly follows the corresponding class definition presented in the previous section.

(make area ∗area_name:seoul ∗population:3000 ∗aerial_photograph:aerial_photograph1 .....
∗display_photograph:display_ aerial_photograph(aerial_photograph))
(make commuter_bus ∗capacity:50 ∗mfg_date:[∗year:89 ∗mon:3 ∗date:25] .....)
<t1>(make station ∗station_name:commuter_bus_depot1 ∗waiting_time:30 .....)
<t2>(make station ∗station_name:commuter_bus_depot2 ∗waiting_time:40 .....)
(make route ∗source_node:<t1> ∗dest_node:<t2> ∗length_of_route:2000 .....)

The last instance, 'route', is a composite object because two of its attributes are associated with two other instances. There are two ways of specifying class type attribute values. One is using object identifiers as shown above. If an identifier is given for an instance, the identifier can be used to declare composite objects. In the above examples, two 'station' instances are first declared, and each is bound to object identifier <t1> and <t2>. Then, these objet identifiers are used as the value of 'source_node' and 'dest_node' in the declaration of the 'route' instance. The other is directly embracing an instance declaration in a composite object, which is illustrated in the next example. The whole instance declaration appears as the value of class type attributes. These two methods of specifying composite objects have no semantic difference.

(make route ∗source_node:(make station ∗station_name:commuter_bus_depot1
∗waiting_time:30 .....) ∗dest_node:(make station ∗station_name:commuter_bus_depot2
∗waiting_time:40 .....) ∗length_of_route:2000 .....)

The capability of declaring composite objects makes it possible to navigate through a number of related objects. Several examples of such composite objects are shown in Figure 2.

4.3 Rules in ICOT

A rule has a condition part and an action part. Unlike an ECA (Event-Condition-Action) rule, events are not explicitly specified. This may facilitate knowledge-based programming since a user can be free from specifying events. It is instead assumed that an event initiating rule interaction is generated whenever the state of WM is changed. Rules can interact by exchanging messages. If some attribute values of an instance satisfy the condition part, then the ICOT inference engine executes the action part. This may change some attribute values, generate output, or create new objects. These may further result in driving some other rules to be eligible to fire. In this section the ICOT syntax of rules is first described. Then, several features of object-based rules, such as a use of object variables, derived class, and inheritance of rules, are discussed.

The ICOT syntax specifying rules is shown below.

```
(defrule rule_name
{[<var>|not](<var>|<class_name> {['!']*<attr_name>':'<attr_val>|<var>}+)}+
=>
{([<command>] <var>|<class_name>{*<attr_name>':'<attr_val>|<var>}+)}+)
<command> = modify | make | remove | write
```

In ICOT, objects are used as the basic unit of inference. When defining a rule, the classes to which the rule is applied need to be specified. This can be done by using a variable or a specific class name. Relevant attributes then follow the class specification. An exclamation mark ('!') can precede an attribute name. This is used to indicate that the rule should be triggered whenever the succeeding attribute value is modified or a new instance is added. The same mechanism is found in [31].

Consider the next two example rules.

```
(defrule compute_total_no_of_vehicles
<f>(commuter_bus_route *length_of_route:<x> *dispatch_interval:<y> *avg_speed:<z>)
=>
(modify <f> *total_no_of_vehicles:(<y>*<z>)/(60*<x>)))


(defrule compute_max_no_of_passengers
<f>(commuter_bus_route *vehicle:<commuter_bus>
    *avg_speed:<a> !*total_no_of_vehicles:<b> *distance:<c>)
(<commuter_bus> *capacity:<k>)
=>
(modify <f> *max_no_of_passengers:(<k>*<a>*<b>)/<c>)))
```

The first rule calculates the total number of vehicles for every instance of the 'commuter_bus_route' class. The calculation is carried out using the length of route, the dispatch interval, and the average speed of an individual commuter bus route. The second computes the maximum number of passengers of a vehicle. This is derived from the average speed, the distance, and the total number of vehicles of a commuter bus route and the capacity of a commuter bus. Notice in the second rule that the 'total_no_of_vehicles' attribute is followed by '!'. If the first rule changes this attribute value, the second one is automatically executed and computes 'max_no_of_passengers'.

In the second rule, '<commuter_bus>', coming just after the 'vehicle' attribute, is an object variable. This binds to an object having the object identifier in the 'commuter_bus' class. So, (<commuter_bus> *capacity:<*k*>), which means that the capacity of a commuter_bus object is *k*, represents a simple navigation in a composition hierarchy link. Dot (·) notation may be used to represent such composition hierarchy links. This notation, however, can sometimes be less convenient than the object variables. Consider a vehicle traveling between a 'source_node', *a*, and a 'dest_node', *b*. If one wants to know the name of the service area and to see a picture of the area, the dot notation may be used in the following way.

```
(defrule inform_sevice_area
(route_network *link_component.source_node:<a> *link_component.dest_node:<b>
    *service_area. area_name:<name> *service_area.aerial_photograph:<image1>)
```

```
=>
(route_network ∗service_area.write:write(<name>)
    ∗service_area.display_photograph:display_aerial_ photograph(<image1>)))
```

The 'service_area' is redundantly specified, which leads to accessing the same object twice. If an object variable is used instead, a single access through the object variable can provide all the information needed. The previous rule can be written by using an object identifier as follows.

```
(defrule inform_sevice_area
(route_network ∗link_component:<route> ∗service_area:<area>)
(<route> ∗source_node:<a> ∗dest_node:<b> )
(<area> ∗area_name:<name> ∗aerial_photograph:<image1>)
=>
(<area> ∗write:write(<name>) ∗display_photograph:display_aerial_photograph(<image1>)))
```

In this example, 'display_aerial_photograph' is a message taking <image1> as its parameter. In general, messages invoke their corresponding methods. If a method generating an object identifier is defined, then the message calling it may involve a composition hierarchy link.

Consider the case that two or more routes have an identical source node and an identical destination node, but not intermediate stations. To deal with this case, a universal quantifier, such as "each" or "all", may have to be introduced. Unfortunately, these quantifiers are unsupported in C-ECLPS, and, hence, in ICOT.

A rule defined for a class can be inherited into the subclasses. In other words, if a particular class name is assigned to a pattern, the rule applies to not only the class but also its subclasses. Consider the 'rushhour_overtraffic' rule shown below.

```
(defrule rushhour_overtraffic
<f>(route ∗no_of_passengers:<x> ∗traffic_portion:<y> !∗max_no_of_passengers:<z>)
=>
(modify <f> ∗overtraffic:<x>∗<y>  -  <z>))
```

This rule computes 'overtraffic,' the number of passengers that exceeds the maximal transportation capacity of the route. Once this rule is specified, it is not required to define the same rule for the 'commuter_bus_route' and 'subway_route' classes. Since these two classes are the subclasses of the 'route' class, they can inherit the rule defined for 'route'. This rule inheritance provides the advantage of code reusability.

Finally, a rule can create a new class which is called a derived class. A derived class is basically a subclass of an object class. Consider the following 'commuter_bus_subway' rule. This rule is activated when a commuter bus route and a subway route have the same source and destination nodes. Notice that the action part of the rule generates a new class, 'commuter_bus_subway_route'. This derived class contains a special attribute, that is 'difference_of_passengers'. The attribute is used to specify the difference between the maximum number of passengers of the commuter bus route and that of the subway route. The class name of such a derived class begins with 'derived_'.

```
(defrule commuter_bus_subway
```

```
(commuter_bus_route ∗source_node:<x> ∗dest_node:<y> ∗max_no_of_passengers:<t>)
(subway_route ∗source_node:<x> ∗dest_node:<y> ∗max_no_of_passengers:<s>)
=>
(make derived_commuter_bus_subway_route ∗source_node:<x> ∗dest_node:<y>
    ∗difference_of_ passengers:abs(<t>, <s>))
```

If instances of a derived class are frequently used, the class can be pre-defined. This can lessen the burden of accessing the 'commuter_bus_route' and the 'subway_route' classes. Otherwise, both classes should always be accessed whenever needed. A derived class, once created, is used independently from its base class. This means that any later modifications of the base class is not mirrored on the derived class.

5. Extension to Fuzzy Inference

ICOT provides a fuzzy inference tool to deal with uncertain information. The declaration of objects and rules explained in the previous section are modified to represent fuzziness. The tool adopts the concept employed in Fprolog [32] in the context of a hybrid object-oriented environment. The Rete match algorithm for object inference is further modified to perform fuzzy inference. A method of inexact matching based on fuzzy theory is utilized. Evaluation methods associated with fuzzy operators and inference units are provided.

5.1 Representation of fuzzy objects

The basic syntax to define classes and instances is extended to represent fuzzy information. Three types of fuzzy information are considered. First, fuzzy type attributes are used to specify uncertain attribute values. Such an attribute is allowed to have a linguistic fuzzy value, like 'crowd', 'long', 'high', and 'old'. Second, it is sometimes uncertain if an instance belongs to a class. An additional attribute, '∗poss', is appended to the attribute list to indicate the membership degree of the instance. This attribute is obviously a real type. Third, fuzzy hierarchical relationships can be optionally specified. A fuzzy *isa* relationship is specified by isa[/real]. This represents the uncertainty involved in generalization hierarchy. Class composition hierarchy is also fuzzified. This fuzzy relationship is specified by [/real] which follows the declaration of an embraced instance. All these syntactic elements to represent fuzzy objects are presented below.

```
(defclass <class_name> isa[/<real>] <class_name> { ∗<attr_name> ':' <f_attr_type> }+
    <possibility>)
[<var>] (make <class_name> {∗<attr_name> ':' <fuzzy_attr_val> | (make <class_name>
    {∗<attr_name> ':'  <fuzzy_ attr_val>) }+ <possibility>)[/<real]>}+ <possibility>)
<possibility> = '∗poss :' <real>

(defclass route ..... ∗length_of_route:f_length_of_route
    *no_of_passengers:f_no_of_passengers ..... ∗poss:real)
(make route ..... ∗length_of_route:long ∗no_of_passengers:crowd ..... ∗poss:0.98)

(defclass commuter_bus_route isa/0.9 route ∗vehicle:commuter_bus ∗total_no_of_vehicles:int
    ∗max_no_of passengers:int ∗default_bus_type:char_city_bus
    ∗temp_vehicle:(commuter_bus)avail_vehicle(length_of_route) ∗poss:real)
```

```
<sr>(make commuter_bus_route ∗source_node:<t1> ∗dest_node:<t2>
    ∗length_of_route:2500 ..... ∗overtraffic:200 ..... ∗vehicle:(make commuter_bus .....
    ∗mfg_date:[∗year:83 ∗mon:05 ∗date:10] ..... ∗poss:0.9)/0.85 ..... ∗poss:0.8)
```

In the 'route' class, 'length_of_route' and 'no_of_passengers' are fuzzy type attributes. When declaring an instance of this class, the attributes have linguistic fuzzy values, such as 'long' and 'crowd'. This is shown in the succeeding instance declaration. This example also shows the membership degree (0.98) indicating that the 'route' instance is contained in the class. A fuzzy *isa* relationship is shown in the declaration of the 'commuter_bus_route' class. This class is specified as a subclass of 'route' with the degree of 0.9. A fuzzy composition is presented in the last instance. The composition link for composite attribute 'vehicle' is fuzzified here with the degree of 0.85. Along with the fuzzified *isa* hierarchies, it differentiates ICOT from Fprolog and fuzzy CLIPS. The inference engine of ICOT is able to support all these fuzzy information.

In order to evaluate such fuzzy values as 'crowd' and 'long', it is required to provide a membership function for each fuzzy value. A membership function can be specified by the mfct declaration shown below. Several examples following the mfct syntax are also listed.

```
(mfct <fuzzy_attr_val> '('<parameter>')'  '='  '(' <expression> ')' )

(mfct crowd(x) = (1+1/1000×|400-x|×2)^-1)
(mfct long(x) = (1+1/1000×|2000-x|)^-1)
(mfct high(x) = (x) )
(mfct old(x) = (1+1/30×|20-(94-x)|)^-1)
```

A fuzzy value can be associated with fuzzy modifiers, such as 'very', 'more or less', 'not', and so on. Such modifiers are generally taken into account by adjusting membership functions as follows.

"very" : μ (very x) = (μ (x))^2
"more or less" : μ (more or less x) = (μ (x))$^{1/2}$
"not" : μ (not x) = 1- μ (x)

5.2 Fuzzy inference

In order to inference with fuzzy information, a fuzzy operator and an inference unit need to be first set up. Various fuzzy operators have been proposed to support the fuzzy union (∪) and the fuzzy intersection (∩). These include min-max (the standard fuzzy operator), probabilistic sum-product, and bounded sum-product [33]. Besides these operators, a user can define his or her own operators. An inference unit in fuzzy matching can be either pattern or attribute. A user can choose one of these two. The default inference unit is pattern. The f_op syntax shown below selects a fuzzy operator to be used, def_op creates a user-defined operator, and infer chooses an inference unit.

```
(set f_op <operator_name> )
    <operator_name> =       min-max
                            | probabilistic sum-product
                            | bounded sum-product
                            | ....
```

(def_op <operator_name> = <expression>)

(set infer <infer_method>)
<infer_method> = pattern | attribute

The meaning of the min-max operator, which is the default operator, is shown below. *A* and *B* are fuzzy sets, and μ_A and μ_B are the membership functions of *A* and *B*, respectively.

$\mu_{A \cup B}(x) = \max [\mu_A(x), \mu_B(x)]$
$\mu_{A \cap B}(x) = \min [\mu_A(x), \mu_B(x)]$

When the condition part of a rule is satisfied, the ICOT inference engine executes the action part. However, since objects handled by the rule are now fuzzified, the truth value of the fuzzy information is not simply true/false, but a degree in-between 0 and 1. Therefore, the inference process needs to consider the degree of pattern matching. For this fuzzy pattern matching, a threshold value is specified in the condition part of a rule. The threshold is called an α - cut value. If the matching degree of a pattern (LHS) is greater than α, then the LHS of the rule is regarded as being satisfied and the RHS can be executed. The fuzzy rule syntax is presented below.

(defrule <rule_name>
{[<var>|not](<var>|<class_name> {['!']∗<attr_name>':'<fuzzy_attr_val>|<var>}+)}+
[('with:' < α >)]
=>
{([<command>] <var>|<class_name>{∗<attr_name>':'<fuzzy_attr_val>|<var>}+)}+)
<command> = modify | make | remove | write

Consider the following fuzzy rule that generates an instance in a derived class 'route_of_prior_increase'. The default inference unit, pattern, is assumed. The other inference unit, attribute, is described later. This rule states that, for an instance that belongs to the 'commuter_bus_route' class, if its overtraffic exceeds 100, its length of route is longer than 1,500, and the vehicle is manufactured before 1985, then the instance is included in the derived class.

(defrule find_route_of_prior_increase
(commuter_bus_route ∗source_node:<s> ∗dest_node:<e> ∗overtraffic:>100
∗length_of_route:>1500 ∗vehicle:<commuter_bus>)
(<commuter_bus> ∗mfg_date:[∗year:<85])
(with:0.6)
=>
(make derived_route_of_prior_increase ∗source_node:<s> ∗dest_node:<e>))

Consider the next instance with the above rule. The possibility that this instance matches with the first pattern of the rule is 0.8, and that with the second is min(0.9, 0.85). Therefore, the overall possibility that the instance matches with the whole pattern is min(0.8, min(0.9, 0.85)) = 0.8. Since this value is greater than the α value, 0.6, the RHS is

executed. It results in creating a new instance and inserting it in the 'derived_route_of_prior_increase' class. The membership value of this new instance is 0.8.

(make commuter_bus_route ∗source_node:<t1> ∗dest_node:<t2> ∗length_of_route:2500 ..... ∗overtraffic:200 ..... ∗vehicle:(make commuter_bus ..... ∗mfg_date:[∗year:83 ∗mon:05 ∗date:10] ..... ∗poss:0.9)/0.85 ..... ∗poss:0.8)

In the following examples, attribute is used as the inference unit.

```
(defrule find_route_of_prior_increase
(commuter_bus_route ∗source_node:<s> ∗dest_node:<e> ∗overtraffic:very_crowd ∗length_of_
    route:long ∗vehicle:<commuter_bus>)
(<commuter_bus> ∗mfg_date:[∗year:old])
 (with:0.6)
=>
(make derived_route_of_prior_increase ∗source_node:<s> ∗dest_node:<e> ))

(defrule find_route_of_prior_increase
(commuter_bus_route ∗source_node:<s> ∗dest_node:<e> ∗traffic_portion:high
    ∗degree_of_congestion: high)
 (with:0.6)
=>
(make derived_route_of_prior_increase ∗source_node:<s> ∗dest_node:<e> ))
```

The rules find routes whose overtraffic is 'very_crowd', length of route is 'long', and vehicles are 'old', or whose traffic portion is 'high' and degree of congestion is 'high'. These fuzzy values can be evaluated using fuzzy membership functions defined by mfct. Examine the next instance with the above rules.

(make commuter_bus_route ∗source_node:<t1> ∗dest_node:<t2> ∗length_of_route:1500 ..... ∗overtraffic:300 ∗traffic_portion:0.7 ∗degree_of_congestion:0.8 ..... ∗vehicle:(make commuter_bus ..... ∗mfg_date: [∗year:83 ∗mon:05 ∗date:10] ..... ∗poss:0.9)/0.85 ..... ∗poss:0.8)

The possibility that the instance matches with the condition part of the first rule is first evaluated. Fuzzy attribute values and fuzzy modifiers are calculated using the fuzzy membership functions and the adjusted fuzzy membership functions described earlier. For example, the degree that the fuzzy attribute, '∗overtraffic:very_crowd', matches with '∗overtraffic:300' is $(\text{crowd}(300))^2 = (1+1/1000 \times |400-300| \times 2)^{-1})^2 = 0.69$. Similarly, the degree for 'length_of_route' is long(1500) = 0.67. Thus, the possibility that the instance matches with the first pattern of the first rule is min(0.69, 0.67) = 0.67. The possibility that the instance matches with the second pattern is old(83) = 0.77. Since these two patterns have a conjunctive (and) relation, the possibility that the instance matches with the first rule takes the minimum of the two possibility values, which is min(0.67, 0.77) = 0.67. The possibility that the instance matches with the second rule is evaluated in the same way, that is min(0.7, 0.8) = 0.7. When several rules are in a disjunctive (or) relation, the one having the greatest possibility is chosen. In this example, the possibility with the second rule is greater than that with the first. Since the resulting possibility, max(0.7, 0.67) = 0.7, is

greater than the threshold value of the rule (0.6), the rule is executed and creates an instance. This possibility, 0.7, becomes the membership degree of the new instance in the derived class. Since the inference unit is attribute, the '∗poss' values are ignored. Figure 3 summarizes the overall process of evaluating the rules.

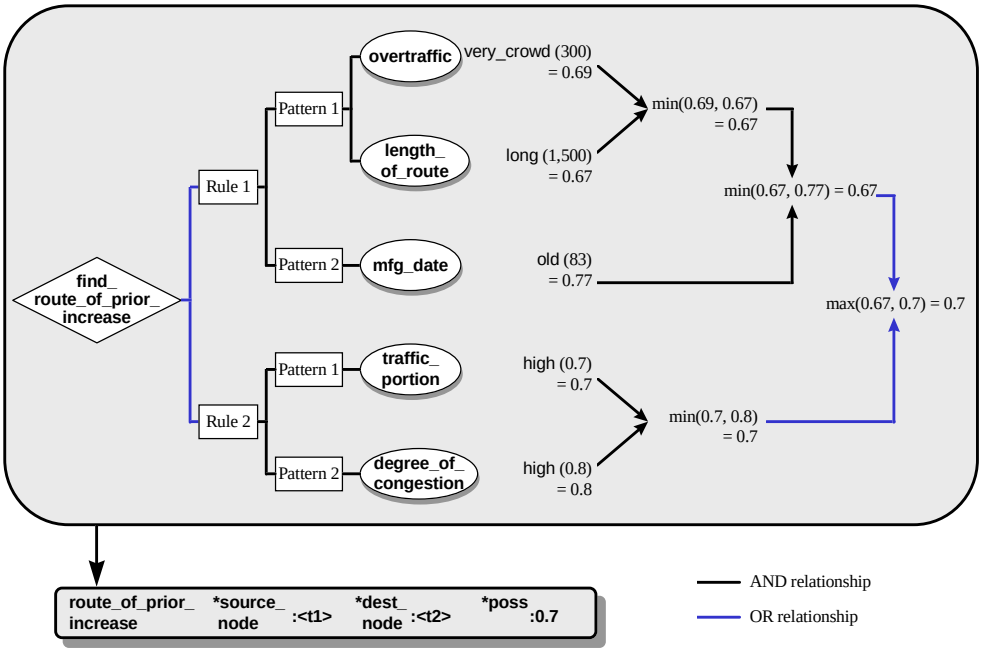

Figure 3. Inference of Uncertainty by Attribute Unit

Now, consider the behavioral semantic of fuzzy *isa* relationship. To illustrate the behavioral semantic, consider the next examples of a class, an instance, and a rule.

```
(defclass subway_route isa/0.9 route ∗name:char ∗default_subway_type:char_subway ∗poss:real)
<sr>(make subway_route ∗source_node:<t1> ∗dest_node:<t2> ..... ∗overtraffic:200 .....
    ∗poss:0.95)

(defrule rushhour_overtraffic
<f>(route ∗no_of_passengers:<x> ∗traffic_portion:<y> !∗max_no_of_passengers:<z>)
 (with:0.85)
=>
(modify <f> ∗overtraffic:<x>∗<y>  -  <z>))
```

Since 'subway_route' is a subclass of 'route', a rule defined for 'route' can also be applied to 'subway_route'. The inference engine first evaluates the possibility that the 'subway_route' instance matches with the condition part of the example rule. Notice that the membership degree of the instance in the 'subway_route' class is 0.95 and the degree that the 'subway_route' class *isa* the 'route' class is 0.9. Since the rule is defined for 'route', we need to know the membership degree of the instance in the 'route' class. This degree, however, is not explicitly specified. Let $\mu_A(<x>)$ denote the membership degree that

instance *x* belongs to class *A*. Then, the possibility that the instance matches with the condition part of the rule can be expressed as min (0.90, μ_{route} (<sr>). It is obvious that μ_{route}(<sr>) $\geq$ μ_{subway_route}(<sr>) since <sr> should first be a route for it to be a subway_route. Therefore, the possibility becomes min (0.90, μ_{subway_route}(<sr>) = 0.95) = 0.90. Since this value is greater than α, the rule can be executed.

6. Conclusions

Proposed in this paper is a composite object inference model that combines a rule-based deductive mechanism with an object-oriented paradigm and fuzzy logic. With the advancement of computer technology, many applications are widely used now to aid complex engineering tasks, such as GIS, scientific, and multimedia operations. The performance of such applications can be greatly enhanced if a knowledge-based tool can support them. However, many currently available rule-based expert system languages have limitations in doing this mainly because the tasks involved in such applications usually need to deal with composite objects and uncertain knowledge. This leads us to the development of ICOT, which is a new knowledge-based programming tool.

ICOT provides a new framework into which rule-based deduction, object-oriented modeling, and fuzzy inferencing are combined altogether. The specification syntax to define classes, instances, and rules is described with examples. Several concepts associated with composite objects, generalization hierarchy, and composition hierarchy have been addressed. Three types of fuzzy information are identified, and a proper way of representing and inferencing them is developed. The tool has been implemented on top of an existing production system language, C-ECLPS. It can be used as a development tool of knowledge-based application software that runs on a relational DBMS. ICOT provides several important benefits. First, the object-oriented concept improves the modeling power of traditional rule-base expert systems. Secondly, it can support various non-traditional data types, such as set, list, tuple, multimedia, spatial, etc. Thirdly, it provides a way to effectively represent the vagueness of the real world. These advantages become especially important for engineering applications. A more sophisticated system may be realized if it can be extended to handle universal quantifiers and equipped with an event driven active mechanism for ECA rules.

References

1. O. R. L. SHENG and C. P. WEI, “Object-oriented modeling and design of coupled knowledge-base/database systems”, In Proceedings of the International Conference on Data Engineering, IEEE, eds. N. J. Cercone (Tempe, Arizona, USA, 1992) pp. 98-105.
2. S. GRECO and P. RULLO, “Complex-prolog: a logic database language for handling complex objects. Information System”, **14**, 1 (1989) 79-87.
3. D. SRIVASTAVA, R. RAMAKRISHNAN, P. SESHADRI and S. SUDARSHAN, “Coral ++ : adding object-orientation to a logic database language”, In Proceedings of the 19th VLDB Conference, eds. R. Agrawals (Dublin, Ireland, 1993) pp. 158-170.

4. P. DECHAMBOUX and C. RONCANCIO, “Peplomd : an object-oriented database programming language extended with deductive capabilities”, In Proceeding of Database and Expert Systems Applications, eds. D. Karagiannis (Athens, Greece, 1994) pp. 2-14.
5. A. M. ALASHQUR, S. Y. W. SU and H. LAM, “A rule-based language for deductive object-oriented databases”, In Proceedings International Conference on Data Engineering, eds. J. Urban (Los Angeles, California, USA) 1990, pp. 58-67.
6. K. V. S. V. N. RAJU and A. K. MAJUMDAR, “Fuzzy functional dependencies and lossless join decomposition of fuzzy relational database systems”, ACM TODS, **13**, 2 (June 1988) 129-166.
7. E. F.CODD, “Extending the database relational model to capture more meaning”, ACM Transaction on Database System, **4**, 4 (1979) 397-433.
8. W. JR. LIPSKI, “On semantic issues connected with incomplete information database”, ACM TODS, **4**, 3 (September 1979) 262-296.
9. J. D. YANG, “Uniform framework for deductive processing of fuzzy information. Fuzzy Sets and Systems”, **64**, 3 (1994) 377-385.
10. L. A. ZADEH, “The role of fuzzy logic in the management of uncertainty in expert systems. Fuzzy Sets and Systems”, **11**, 3 (1983) 199-227.
11. J. J. BUCKLEY, W. SILER and D. TUCKER, “A fuzzy expert system”, Fuzzy Sets and Systems, **20**, 1 (1986) 1-16.
12. K. S. LEUNG and W. LAM, “Fuzzy concepts in expert systems”, IEEE Computer, **21**, 9 (September 1988) 43-56.
13. Y. M. SHYY and S. Y. W. SU, “K: A high-level knowledge base programming language for advanced database applications”, In Proceeding of ACM SIGMOD, eds. James Clifford and Roger King (Denver, Colorado, USA May, 1991) pp. 29-31,.
14. J. D. YANG, “F_MP: A fuzzy match framework for rule-based programming. Data and Knowledge Engineering”, to appear (1997).
15. M. DALAL and D. GANGOPADHYAY, “OOLP: A translation approach to object-oriented logic programming”, In Proceedings of the First International Conference on Deductive Object-Oriented Database, eds. W. Kim (Kyoto, Japan, 1989) pp. 555-568.
16. K. FUKUNAGA and S. HIROSE, “An Experience with a Prolog-based object-oriented language”, In OOPSLA '86 Proceedings, eds. N. K. Meyrowitz (Portland, Oregon, USA, September 1986) pp. 224-231
17. C. ZANIOLO, “Object-oriented programming in Prolog”, In Proceedings of IEEE International Symposium on Logic Programming, (Atlantic City, USA, February 1984) pp. 265-270.
18. K. LEE and S. LEE, “An object-oriented approach to data/knowledge modeling based on logic”, In Proceeding of International Conference on Data Engineering, eds. J. Urban, IEEE, (Los Angeles, California, USA, February, 1990) pp. 289-294.
19. M. STONEBRAKER, “Object-relational DBMS-the next wave”. An Illustra Technical White Paper, Illustra Information Technologies, Inc., (1995).
20. U. DAYAL, A. P. BUCHMANN and D. R. MACARTHY, “Rules Are Object Too: A Knowledge Model For An Active, Object-Oriented Database System”, In Proceedings of the Second International Workshop on Object-Oriented Database System (Ebernburg, Germany 1988) pp. 129-143.

21. D. MAIER, “A logic for objects”, In Proceeding of the Workshop on Foundation of Deductive Database and Logic Programming (1986) pp. 6-26.
22. J. C. GIARRATANO, “CLIPS user's guide”, Artificial Intelligent Section, NASA/Johnson Space Center (1987).
23. S. Y. W. SU and H. X. LAM, “An object-oriented knowledge base management for supporting advanced applications”, In Proceeding of 4th International HK Computer Society Database Workshop (1992).
24. C. L. FORGY, OPS5 user’s manual, Department of Computer Science, Carnegie-Mellon University (1994).
25. R. A. ORCHARD, FuzzyCLIPS version 6.02: a user’s guide. Knowledge Systems Laboratory: Institute for Information Technology, National Research Council, Canada, September (1994).
26. Y. H. LIM, S. J. YOO, J. S. LEE and S. C. BAEK, CRS : user manual of the expert system building tool (I). ETRI, ETRI Internal Report No.TD91-6220, (1991).
27. M. I. SCHOR, T. P. DALY, H. S. LEE and B. R. TIBBITTS, “Advances in Rete pattern matching” In Proceeding of AAAI - 86 (Philadelphia, Pennsylvania, USA, 1986) pp. 226-232.
28. BELL ATLANTIC KNOWLEDGE SYSTEM. The laser environment. Bell Atlantic Knowledge System (1989).
29. UniSQL, Inc. UniSQL/XTM database management system user's manual. Release 1.2, UniSQL, Inc. (1992).
30. O. DEUX, “The story of O2”, IEEE Transactions on Knowledge and Data Engineering, **2**, 1, (March 1990) 91-108.
31. IBM, Enhanced common LISP production system user's guide and reference. First Edition, International Business Machine Corporation (1988).
32. T. P. MARTIN, J. F. BALDWIN and B. W. PILSWORTH, “The implementation of Fprolog -a fuzzy prolog interpreter”, Fuzzy Sets and Systems, **23**, 1 (1987) 119-129.
33. H. J. ZIMMERMANN, Fuzzy set theory and its applications. (2nd ed.), Kluwer-Nijhoff Publishing (1991).