AN OBJECT-ORIENTED MODULAR EXPERT SYSTEM SHELL

By

JIN CHEON NA

Bachelor of Science in Electrical Engineering

Hanyang University

Seoul, Korea

1987

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1990

AN OBJECT-ORIENTED MODULAR EXPERT

SYSTEM SHELL

Thesis Approved:

Blayne E. Mayfield

Thesis Advisor

J P Chandler

Norman N. Durham

Dean of the Graduate College

ACKNOWLEDGMENTS

I want to show my most heartfelt appreciation to my major advisor Dr. Blayne E. Mayfield, for his warm encouragement and guidance on this study. I also like to extend my thanks to Dr. John P. Chandler and Dr. K. M. George for their advise and close reading of the thesis, and for serving as members of my graduate committee.

I would like to express my appreciation to Dr. Eui-Ho Suh for serving as my major advisor in the beginning of this study. I also like to express my thanks to my parents, Joong Bae Na, and Young Keun Na for their love and support during my thesis writing.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

OOMESS (an Object-Oriented Modular Expert System Shell) is an expert system shell which integrates different knowledge representation schemes into an object-oriented programming environment. This thesis describes the design issues of OOMESS. For this thesis, the scope of OOMESS is limited to backward-chaining inference strategy. OOMESS provides interface from rules to access the underlying object-oriented language, which provides the environment for OOMESS, and to access rule group objects. Rule group objects will be described in Chapter IV. One of the motivating forces in the design of OOMESS is a desire to provide system support for the modularization of a large rule base into smaller rule groups and to provide flexible interactions among rule groups.

OOMESS supports object-oriented design as described below. To develop an expert system (i.e., to solve a problem) in the OOMESS environment, the problem to be solved is divided into smaller problems according to the level of detail of the problem. An object-oriented development concept can be used to divide a problem into separate smaller problems [BOOC86], for this concept offers

a mechanism that captures a model of the real world [BOOC86]. The knowledge base necessary to solve the problem can also be partitioned into separate smaller ones. These smaller knowledge bases can be encapsulated into user-defined objects and Expert System Shell (ESS) objects which are instances of the Expert System Shell (ESS) class. User-defined objects can be used for storing more complex knowledge that is not neatly encapsulated in the form of production rules. ESS objects are used to store rule groups. ESS objects and user-defined objects are expected to have the properties of data abstraction [LISK75] and information hiding [PARN72] because these objects are defined within the object-oriented language.

Problem solving is performed through message passing among ESS objects. This message passing provides very flexible interactions among rule group objects (i.e., ESS objects) since one rule group object can send a message to any of the other rule group objects. Also, the OOMESS design does not prohibit recursive message passing.

OOMESS, however, does not possess all the properties of an object-oriented system in the commonly accepted sense because it has neither class hierarchy nor inheritance properties among ESS objects (i.e., every ESS object is an instance of the ESS class). But OOMESS is designed to provide a hierarchical relationship among ESS objects, which gives children ESS objects the privilege of access to the Local Working Memory of a parent ESS object. When an

ESS object searches for a value in the Local Working Memory of its parent ESS object, the parent ESS object attempts to infer the value if it is unknown. An ESS object searches the Local Working Memory of its parent ESS object when the variables it is looking for are not declared in its Local Working Memory. This process is called "Local Working Memory Inheritance." The details of this feature will be described in Chapter IV. Except for ESS objects, every object, defined in the object-oriented programming environment of OOMESS, has both class hierarchy and inheritance properties.

The resulting system is one in which a library of ESS objects (relevant to different problem solving tasks) and user-defined objects are available. During a consultation, one ESS object is selected by the user, and the other ESS objects and/or user-defined objects are selected automatically by the system through the goal-directed nature of the system using the support provided by the environment. This technique should provide an efficient mechanism for managing large rule sets and voluminous working memories since a large rule base and a large working memory are modularized to smaller ones respectively. Furthermore, this approach supports reusability, which is a novel concept in expert system development.

CHAPTER II

OBJECT-ORIENTED CONCEPTS AND EXPERT SYSTEM SHELLS

Object-Oriented Concepts

Many ideas of object-oriented (O-O) programming were introduced by SIMULA [DAHL66] in which the fundamental notions of objects, messages, and classes were employed. Then the first substantial interactive, display-based implementation appeared: the SMALLTALK language [GOLD83]. In 1982, Rentsch stated that "Object-oriented programming will be in the 1980's what structured programming was in the 1970's" [RENT82]. In 1986, Mark Stefik and Daniel G. Bobrow [STEF86] observed that there were probably fifty or more object-oriented programming languages in use. But different O-O programming languages provide varying degrees of support for the principles of O-O programming. One thing these languages have in common is that they share the concept of objects which are entities that encapsulate the data and the procedures that manipulate the data [STEF86].

Programs written in the O-O programming languages consist of objects that combine the properties of procedures and data since they perform computations and save local data [STEF86]. In most O-O languages, objects

are divided into two major categories: classes and instances [STEF86]. A class is a collection of one or more similar objects called instances [STEF86]. For example, APPLE is a class since it represents the set of all apples of APPLE, and APPLE-1 is an instance (Figure 2-1). Figure 2-1 shows the class-subclass hierarchy in a single inheritance model, which specifies the inheritance relationship among classes using a tree structure. A class that is lower in the class hierarchy than a given class is called a "subclass"; a class that is higher than a given class is called a "superclass." When a class (APPLE, for instance) is placed in the class hierarchy, it "inherits" variables and procedures (called "methods") from its superclasses (FRUIT). This means that an instance of a class can inherit any variable or procedure defined in the parent of the class in the class hierarchy. This process is called "inheritance." For example, APPLE-1 inherits "sweet" from FRUIT. But if a class redefines any variable or procedure that already appears in its superclasses, the redefined variables and procedures in the superclasses are not inherited by this class. For example, the LEMON class does not inherit the value of taste from the FRUIT class because the LEMON class has its own taste variable (Figure 2-1). Inheritance in O-O programming languages enables the easy creation of objects that are almost like other objects with a few incremental changes (if any), and it further enhances a knowledge system's ability to reason [BOOC86].

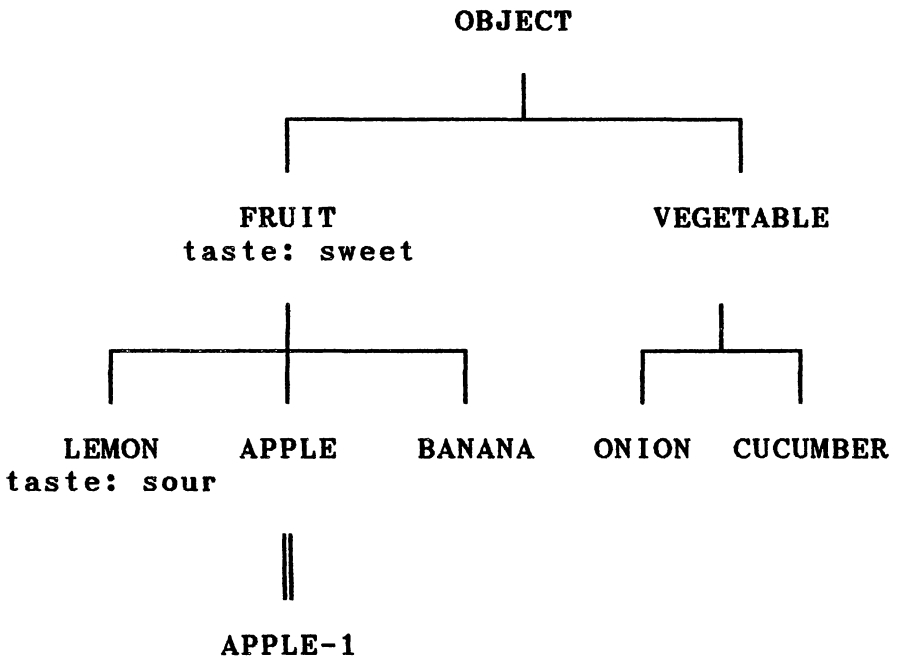


Figure 2-1. The Class-Subclass Hierarchy.

All actions in O-O programming are the consequence of sending messages between objects. As an example, consider the message passing scheme employed in Loops [BOBR83], which is a form of indirect procedure call. Instead of naming a procedure to perform an operation on an object, one sends the object a message. A message consists of arguments and a selector: the selector in the message is used to find the associated method (a procedure to answer to the message) in the message-receiving object [STEF86]. The object receiving a message executes its own methods for performing the required operations. Message sending supports data abstraction: calling objects do not make assumptions about the implementation and internal representations of called objects [STEF86].

O-O programming provides good facilities for simulation programs, graphics, AI programming, and system programming [STEF86]. Recently, O-O programming languages have become popular in the area of artificial intelligence due to their similarity to existing techniques for knowledge representation such as frame [MINS75] and for their use in knowledge acquisition [CASA88]. O-O programming languages also provide an efficient software design method in which the decomposition of a system is based upon the concept of an object [BOOC86]. This software design method is different from the traditional functional decomposition methods used in procedure-oriented programming languages [BOOC86]. According to Booch, the

greatest strength of an object-oriented approach to software development is that it offers a mechanism that captures a model of the real world [BOOC86]. According to Stefik, even though O-O programming languages have a long history, agreement on the fundamental principles of O-O programming is needed for standardizing O-O programming languages [STEF86].

Expert System Shells

Expert systems have been developed since the 1960s, and the first expert system appeared in the early 1970s: an expert system is a computer program or a set of programs capable of performing near, at, or above the level of a human specialist solving problems in a narrow domain [SAUE83]. The most significant development in the mid-1980s has been the proliferation of expert system-building tools and environments that assist expert system builders and the users of those expert systems. Any set of software tools that assists expert system builders in the development of an expert system beyond programming languages such as LISP, PROLOG, and SMALLTALK is called an expert system shell [MART88].

An expert system uses knowledge, facts, and reasoning techniques to solve problems that normally require human specialists for their solution [MART88]. The system knows a great deal about a specific domain of knowledge rather than a general one. This characteristic is called "highly

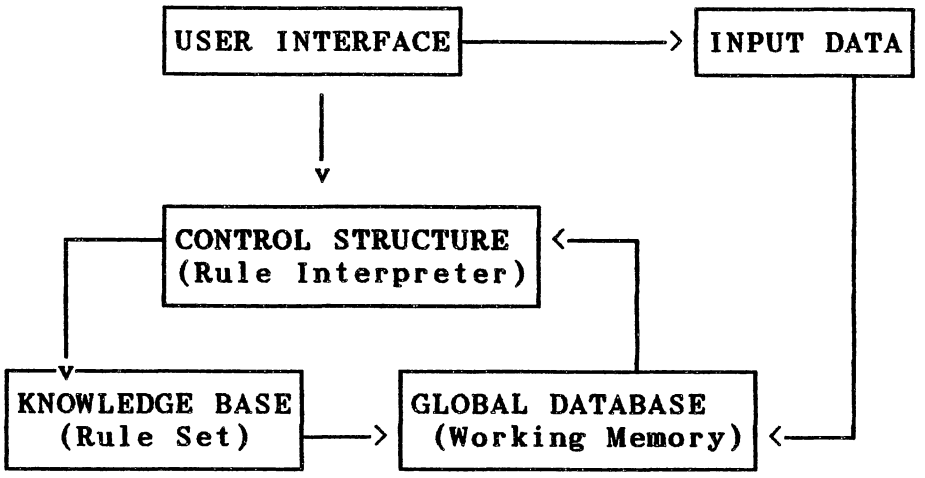


Figure 2-2. Components of an Expert System [FIRE88, p. 337].

domain specific."

Most production-rule-based expert systems include the basic components as shown in Figure 2-2 [FIRE88, p. 337]. Production rules [DAVI77], condition/action decision rules, are normally structured in the form of "IF conditions THEN actions." If the condition part of a rule is satisfied, the rule becomes applicable and the action part is executed. Production rules have been the most effective knowledge representation for declarative specifications of domain-dependent behavioral knowledge, and are easily understood by domain experts [FIKE85]. A user interface ranges from a simple menu-driven I/O to sophisticated natural language dialogues and commands. A knowledge base is generally structured in the form of production rules. An Inference engine (i.e., the control structure in Figure

2-2) applies the knowledge base information (i.e., rule set and working memory) for solving problems. The current problem status is stored in the working memory.

Many expert system control strategies are in use today. Some of the more popular control strategies are backward-chaining, forward-chaining, breadth-first search, depth-first search, heuristic search, problem reduction, pattern matching, hierarchical control, unification, and event-driven control [MART88, FIRE88]. One or more of these strategies is incorporated into the inference engine of each expert system shell. Backward-chaining and forward-chaining are strategies used to specify how rules are to be executed.

As an example of control strategy, the depth-first search strategy with the backward-chaining paradigm is considered. Let us consider the following rules [MART88, p. 241]:

```
rule1: IF weather is sunny
       AND distance <= 20 miles
       THEN transportation is bicycle.

rule2: IF transportation is bicycle
       THEN no passenger insurance is considered.

rule3: IF no passenger insurance is considered
       THEN transportation insurance cost = 0.

rule4: IF no insurance company exists
       THEN it is impossible to get insurance.

rule5: IF it is impossible to get insurance
       THEN transportation insurance cost = 0.
```

When the goal is "transportation insurance cost," the

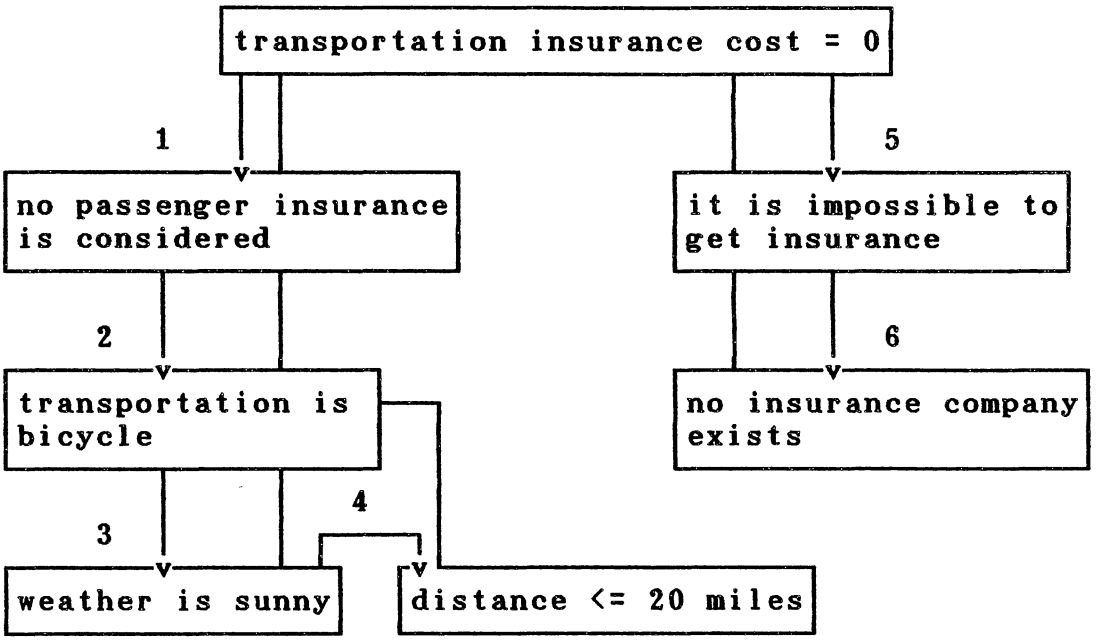


Figure 2-3. Depth-First Search Strategy together with Backward-Chaining paradigm [MART88, p. 241]

process of achieving the goal is illustrated in Figure 2-3 [MART88, p. 241]: the numbers in the figure specify the order in which the goal is considered.

Although expert system technology is not the ultimate technology compared with the other aspects of AI technology, it is the most significant practical product to emerge from 30 years of AI research [FIRE88]. The following areas--knowledge representation, knowledge acquisition, expert system tools, expert system design and expert system programming--demand further research to improve expert system technology [MART88, PARS88, SIEG86].

CHAPTER III

REVIEW OF CURRENT EXPERT SYSTEM SHELLS

Introduction

Any set of software tools designed to assist expert system builders in the development of expert systems beyond programming languages (like LISP, PROLOG, and SMALLTALK) can be called an expert system shell [MART88]. In particular, any expert system building tool that is designed to be used by knowledge engineers in the construction of expert systems should be considered an expert system shell. The basic components of expert system shells are inference engine, user interface, explanation facility, and knowledge acquisition facility [MART88].

Some sets of expert system building tools offer choices of knowledge representation methods and inference strategies. Therefore, expert system builders can select a particular knowledge representation scheme and inference strategy. These tools are not called expert system shells, but expert system programming environments [MART88]. In this paper we use the term "expert system shell" to mean any collection of expert system building tools. The following sections describe several commercial expert system shells that are related to OOMESS.

Personal Consultant Plus

Personal Consultant, developed by Texas Instruments, is an expert system shell that helps expert system builders create expert systems that run on personal computers. Personal Consultant Plus, an enhanced version of Personal Consultant, offers an augmented knowledge base capacity, a frame-based knowledge representation feature, and an enhanced graphics program interface that allows graphics and other programs to be used in the rule bases of Personal Consultant Plus applications [MART88, TEXA87].

Personal Consultant Plus uses frames, parameters, and rules to represent knowledge bases. To develop an expert system, to solve a problem, in the Personal Consultant Plus environment, the problem to be solved is divided into smaller problems (i.e., smaller rule bases) according to the level of detail of the problem. Frames [MINS75], which provide structured representations of stereotyped objects or classes of these objects, are used to store these smaller rule bases of the expert system. Rules and parameters are associated with a frame.

Each knowledge base has a root frame and one or more subframes. Figure 3-1 shows the relationships among the frames. The parent-child relationship is obvious from the figure. For example, the root frame A has as children subframes B and C. A root frame captures the most general concepts of a domain knowledge, and more specific concepts

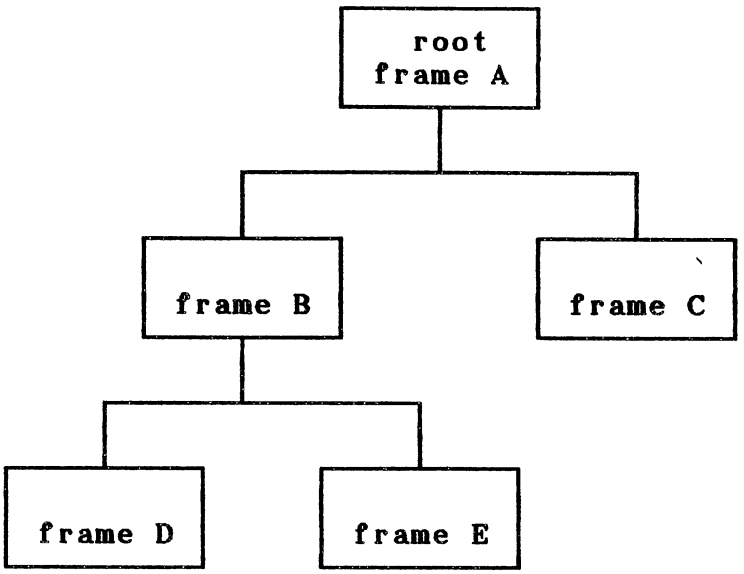


Figure 3-1. An Example of Relationships among Frames [TEXA87].

are grabbed by subframes. Properties associated with frames, parameters, and rules determine their characteristics. Frames, for example, have a property like GOALS. GOALS are lists of parameters whose values should be solved when frames are instantiated.

Parameter groups that consist of logically related parameters can be associated with more than one frame. Parameters are used to store data in frames. The system provides rule groups in which rules are organized. Each frame has a rule group which can be connected with more than one frame. But frames that have identical rule groups must also have the identical parameter groups that correspond to the rule groups.

A rule is structured in the form of an IF-THEN statement. System functions (provided by Personal Consultant Plus) and user-defined functions can be accessed from rules; LISP statements can also be used in rules. The Abbreviated Rule Language (ARL) provides a convenient format for entering system functions required in rules [MART88]. A number of properties are associated with rules. For example, a property, ANTECEDENT, is used to indicate that the rule uses forward-chaining inference strategy. Rules whose ANTECEDENT property is not YES can only be used in the backward-chaining mode.

Personal Consultant Plus provides an inheritance mechanism which applies to parameter groups associated with frames, which means that parameter groups in parent frames are inherited by children frames. However, the inheritance mechanism does not apply to rule groups. The inheritance mechanism applied to the frames in Figure 3-1 is shown in Figure 3-2.

Martin and Oxman [MART88] state that "though speed and memory space continue to be obstacles in building large expert systems on PCs, the increased capabilities offered by Personal Consultant Plus make the task easier and more practicable."

Nexpert Object

Nexpert Object, developed by Neuron Data, is a hybrid rule- and object-based shell operating on IBM mainframes

frame	can access parameters in	can invoke rules in
A	A	A B C D E
B	A B	B D E
C	A C	C
D	A B D	D
E	A B E	E

Figure 3-2. The Access of Each Frame to Rules and Parameters [TEXA87].

under VM, VAXstations under VMS, IBM AT, PS/2, Macintosh Plus, and so forth [ARCI88, NEUR87]. In addition to the usual rule-based method, Nexpert Object uses a knowledge representation paradigm with objects [BROW88, NEUR87]. That is, only part of a knowledge base is codified in the form of rules, and the rest in the form of objects. Because of this methodology, Neuron Data calls its own product an "object-based expert system shell." But these objects do not contain their own rules; Nexpert Object uses rules to reason about these objects and classes.

Rules in Nexpert Object have forward/backward symmetry, which means that the same rules can be used for both forward and backward chaining [BROW88]. A set of rules that share data or hypothesis, forms a knowledge island which divides the knowledge base for fast access to information [BROW88].

Rules are comprised of "If ... Then ... and Do ..." statements, where If is followed by a set of conditions, Then by a hypothesis, and Do by a set of actions to be undertaken if all the conditions become true.

Nexpert Object, which provides a dual mode of knowledge representation and a flexible, graphical interface, is a more powerful and sophisticated system than those commonly found on minicomputers and mainframes [BROW88].

Opus

Opus, an Object-Oriented Production System, is a tool for rule-based programming in the Smalltalk-80 environment which integrates a production system paradigm [LAUR87]. A data-driven production system in Opus allows access to the full functionality of the Smalltalk-80 language; it also allows the ability to match rules with arbitrary objects in the environment [LAUR87]. Laursen [LAUR87] points out that "the design of Opus was driven by the desire for a close integration of production system, language and environment, and for maximum freedom of expression in the rule language."

There are several other production systems written in object-oriented languages. For example, Humble [PIER86] provides an Emycin-like [BUCH84] expert system shell that runs in the Smalltalk-80 language and programming environment; Orient84/K [TOKO85] adds rules and

working memory to Smalltalk classes; and YAPS [ALLE83] supports the use of lisp Flavors in working memory [LAUR87].

KEE

KEE, Knowledge Engineering Environment [KEHL84], has achieved a great deal of success by integrating frame and production rule languages to form hybrid representation facilities that combine the advantages of both languages [FIKE85].

The production rules in the KEE system are represented as frames. This feature allows rules to be grouped into classes and to contain supplementary descriptive information in frame slots. A rule's conditions and conclusions within a frame are represented by a simple predicate logic language. Fikes [FIKE85] argues that "a frame-based representation facility extends the system's explicitly held set of beliefs to a larger, virtual set of beliefs by automatically performing a set of inferences as part of its assertion and retrieval operation."

Fikes [FIKE85] also states that "one of the major advantages of this kind of hybrid facility is that it makes the expressive and organizational power of object-oriented programming available to domain experts who are not programmers."

HAPS

Being successful, expert system technology is applied to broader and more complex domains than before. Therefore, expert system shells must have the capability of handling larger rule bases and a much larger working set; some expert system shells are also required to operate in real-time situations [SAUE83].

These constraints are incorporated into the design of HAPS (the Hierarchical, Augmentable Production System). HAPS is a goal-directed system, which provides the classes of production rules, hierarchical levels of working memory, the dynamic construction of production hierarchies, and modular, modifiable sets of control strategies and conflict resolution strategies [SAUE83]. HAPS also provides additional, globally accessible memory types designed to support the implementation of large expert systems in real-time situation [SAUE83].

LOOPS

LOOPS, developed by Xerox Corporation, is a multiple paradigm system that adds data, object, and rule-oriented programming to the procedure-oriented programming of Interlisp-D which is a dialect of the LISP programming language [BOBR83]. Therefore, a user can choose the style of programming which suits his/her application.

Rules in LOOPS are organized into RuleSets that

consist of an ordered list of rules and a control structure. Therefore, In the rule-oriented paradigm, programs are separated into their RuleSets. The control structures of RuleSets are data-driven inferencing strategies that determine which rules are executed: in LOOPS, there are several different control structures such as Do1, While1, and DoNext [BOBR83]. The rules, defined as classes of LOOPS objects, consist of three parts called the left hand side (LHS) for containing the conditions, the right hand side (RHS) for containing the actions, and the meta-description (MD) for containing the rule descriptions [BOBR83, MART88].

Since LOOPS is integrated into the Interlisp-D environment, it provides access to Lisp programming and the extensive environmental support of the Interlisp-D system [BOBR83].

CHAPTER IV

AN OBJECT-ORIENTED MODULAR EXPERT SYSTEM SHELL

Introduction

OOMESS (an Object-Oriented Modular Expert System Shell) is an expert system shell which is designed to integrate a production system with an object-oriented language. This thesis is concerned with the design of OOMESS. The OOMESS environment is defined on top of an object-oriented language (e.g., C++, Smalltalk-80, Loops) [GUTM89]. An obvious advantage of implementing a rule system within an object-oriented programming language is the opportunity to take advantage of the underlying language and hence to factor the system into modular components [LAUR87]. The modular components are reusable software components which tend to be objects or classes of objects [BOOC86]. Given a rich set of modular components, our implementation proceeds via composition of these components. The OOMESS environment consists of a collection of objects: Expert System Shell (ESS) objects, Global Working Memory, a backward-chaining inference engine object, built-in objects, user-defined objects, and a user interface.

The Expert System Shell (ESS) class is a template for the ESS object. The ESS class consists of two methods (New and Activation) and four instance variables (Inference-Engine-Type, Object-Parameters, Rule Base, and Local Working Memory). Instances of the ESS class, called ESS objects, are used for defining rule groups. Global Working Memory is used for sharing global data among ESS objects. Local Working Memory is visible only to the ESS object in which it is defined. Global working memory elements have the same data structures as local working memory elements. User-defined and built-in objects, which are written in the object-oriented language used for implementing OOMESS, can be accessed from the rules in the Rule Base. Therefore, some portion of a knowledge base can be stored in user-defined and built-in objects. Built-in objects are provided when OOMESS is developed, and user-defined objects can be added by the user (i.e., a knowledge engineer). Normally, these objects are used for storing more complex knowledge which is not neatly encapsulated in the form of production rules. Since built-in and user-defined objects are defined using the object-oriented language, these objects may have class hierarchy and inheritance properties [COX86, MARK86].

The Inference-Engine class may have several different instances (e.g., backward-chaining, forward-chaining, hybrid backward- and forward-chaining inference engine object, etc.). Therefore, all ESS objects do not need to

have the same inference strategy. However, for the sake of simplicity, this thesis assumes that the Inference-Engine class has only a backward-chaining inference engine object as its instance; therefore, all ESS objects adopt a backward-chaining inference strategy uniformly. The Inference-Engine class can be improved to have other inference strategies as its instances in future work.

The role of the user interface in OOMESS is to provide the tools for the user to create expert systems (i.e., ESS objects, Global Working Memory, and user-defined objects), and to perform consultations [HEND88].

One of the anticipated advantages of OOMESS is that when a large knowledge base is developed within OOMESS, it can be divided into smaller knowledge bases according to the level of detail of the problem. These smaller knowledge bases are stored in user-defined objects and ESS objects. Because user-defined objects and ESS objects are defined within the object-oriented language, they have the properties of data abstraction [LISK75] and information hiding [PARN72].

Now we will examine the instance variables and methods of the ESS class. The structure of the Expert System Shell class is shown in Figure 4-1. The Object-Parameters instance variable contains parameters that determine the properties of an ESS object. The Rule Base instance variable contains rules that are structured in the form of production rules. The Local Working Memory instance

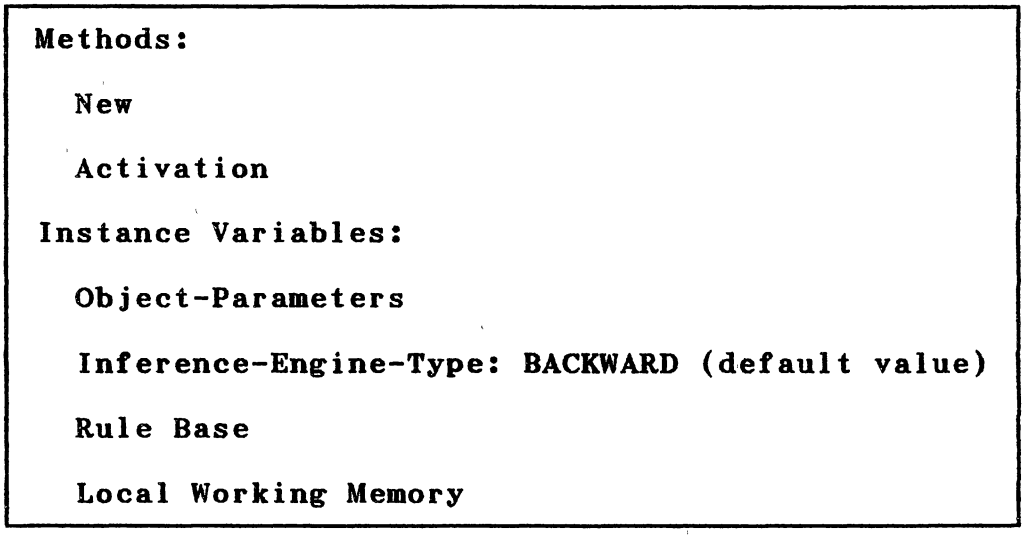


Figure 4-1. Expert System Shell (ESS) Class

variable contains elements which are input by a user or the data inferred in its ESS object as OOMESS executes a consultation. The Inference-Engine-Type instance variable is used for selecting one of the instances of the Inference-Engine class. The Inference-Engine-Type has the default value BACKWARD which is set to the Inference-Engine-Type of every ESS object unless the default value is explicitly changed: the value BACKWARD indicates that the inference strategy used is backward-chaining. These instance variables will be described in detail in the following sections.

The "New" method is used for creating an instance of the ESS class (i.e., an ESS object). A knowledge engineer uses this method to create a new ESS object with its attribute values for Inference-Engine-Type, Object-

Parameters, Rule Base, and Local Working Memory. All ESS objects in an expert system are created when the user develops the expert system. The development processes of expert systems will be described in the User Interface section. The "Activation" method is used for activating an ESS object during a consultation. The "Activation" method in an ESS object invokes the inference engine object (i.e., an instance of Inference-Engine class) specified by the value of the Inference-Engine-Type instance variable, and the invoked inference engine object controls search and inference in the ESS object; this inference engine has the privilege of accessing the instance variables of the ESS object.

Problem solving is performed through message passing among ESS objects (Figure 4-2). Message passing is actually executed from rules in the Rule Base. When one ESS object sends a message to another ESS object, the inference engine object operating on the message-sending

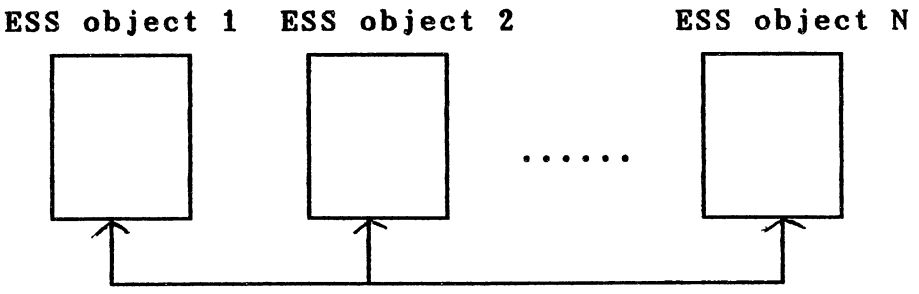


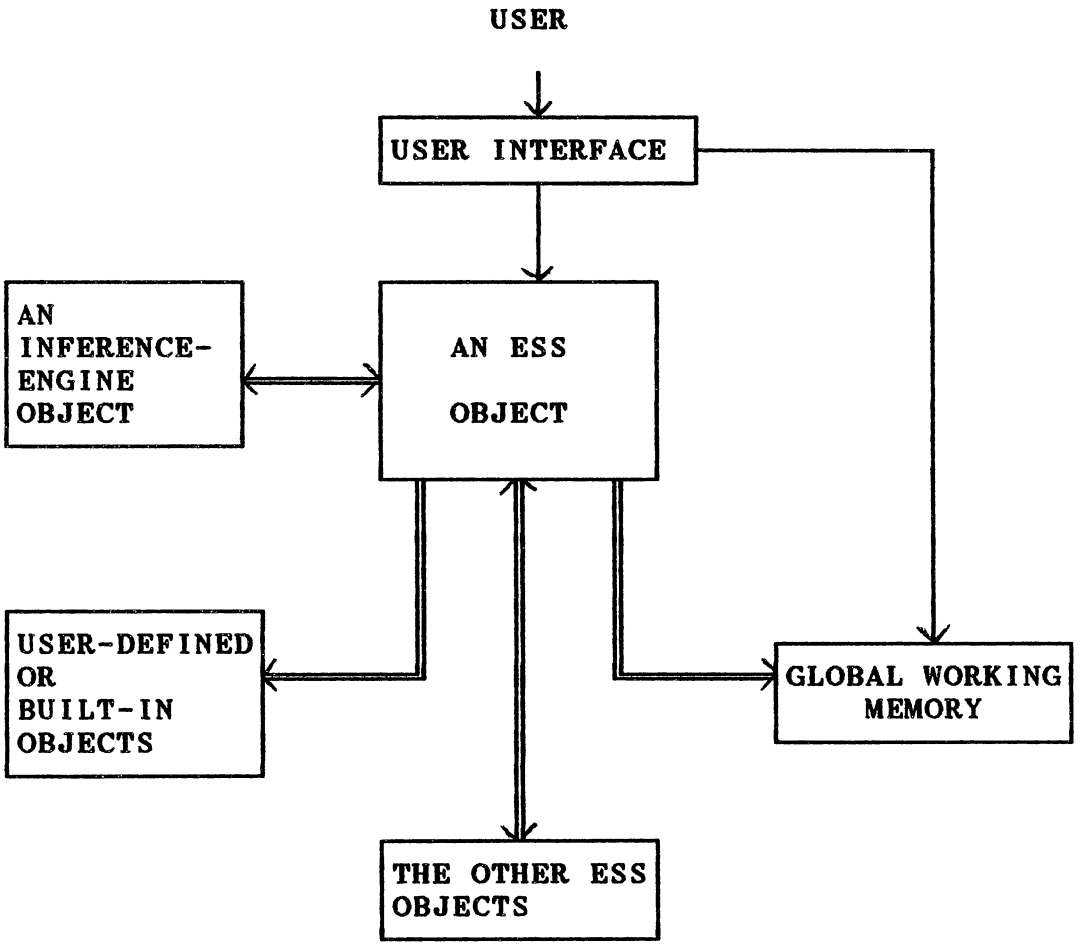
Figure 4-2. Message Passing among Expert System Shell Objects.

object relinquishes control to the activation method of the message-receiving object, and the activation method invokes an inference engine object according to the value of the Inference-Engine-Type.

The general configuration of an object-oriented modular expert system is shown in Figure 4-3. An object-oriented expert system developed under OOMESS consists of any number of ESS and user-defined objects, Global Working Memory, built-in objects, a backward-chaining inference engine object, and a user interface.

As can be seen in Figure 4-3, the rules of an ESS object can send messages to user-defined objects, built-in objects, Global Working Memory, and other ESS objects for solving goal values. The resulting system is one in which a library of ESS objects (relevant to different problem solving tasks) and user-defined objects is available. Several ESS objects and/or user-defined objects are selected during the system execution, and the goal-directed nature of the system guides the search through these selected ESS objects.

To describe OOMESS, we will use a "Travel Assistant Expert System" (which is given in APPENDIX B) as an example in this chapter. The goal of the Travel Assistant Expert System is to help travelers decide the mode of transportation. The expert system consists of the ESS objects TRANSPORTATION, AIRPORT, CAR, CAR-AIR, TRUCK, PASSENGER-CAR.



——— : Data Flow

═══ : Message Passing

Figure 4-3. The General Configuration of an Object-Oriented Modular Expert System.

The Object-Parameters Instance Variable

The Object-Parameters determine the properties of an ESS object. When an ESS object is created, it acquires characteristics according to the values of the Object-Parameters. The Object-Parameters consist of OBJECT_NAME, GOAL_CANDIDATES, PASSED_ARGUMENT, and SUPER_LWMOBJECT. As an example, the Object-Parameters of the ESS object AIRPORT are shown in Figure 4-4.

The value of OBJECT_NAME specifies the name of the ESS object in which it is contained. The GOAL_CANDIDATES property specifies the possible goal parameters that can be solved by the ESS object. Local working memory elements are also called "parameters" in this thesis. When a

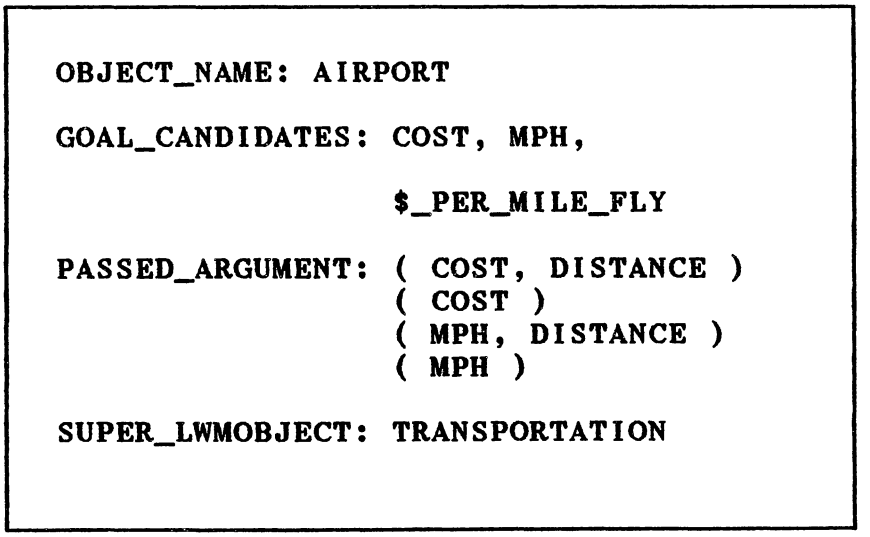


Figure 4-4. An Example of the Properties of an Object-Parameters.

consultation begins, these goal parameters are listed under the name of every ESS object on the screen. Therefore, the user can see the possible goals of every ESS object and select the goal name of a specific ESS object. The objective of the consultation, then, is to find a value for the chosen goal parameter.

The PASSED_ARGUMENT defines the goal names and arguments passed into and out of its ESS object. The first parameter in a PASSED_ARGUMENT list contains a goal name, and there can be any number of passing arguments. For example, in the first PASSED_ARGUMENT list (COST, DISTANCE), the first parameter (COST) is a goal name and the second parameter (DISTANCE) is a passing argument (Figure 4-4). The PASSED_ARGUMENT lists in an ESS object are used when the ESS object receives messages sent by other ESS objects; there is one list for each message. This feature will be described in detail in The Rule Base Instance Variable section. The first element of a list is a goal parameter (refer to Figure 4-4). Only the goals defined in the PASSED_ARGUMENT lists of an ESS object can be requested by other ESS objects. If two goals in an ESS object can be requested from other ESS objects, there will be two PASSED_ARGUMENT lists and so on.

The SUPER_LWMOBJECT specifies the name of a parent ESS object. The parent-child relationship is based on and is restricted to the Local Working Memory. The hierarchical relationship among the ESS objects in the Travel Assistant

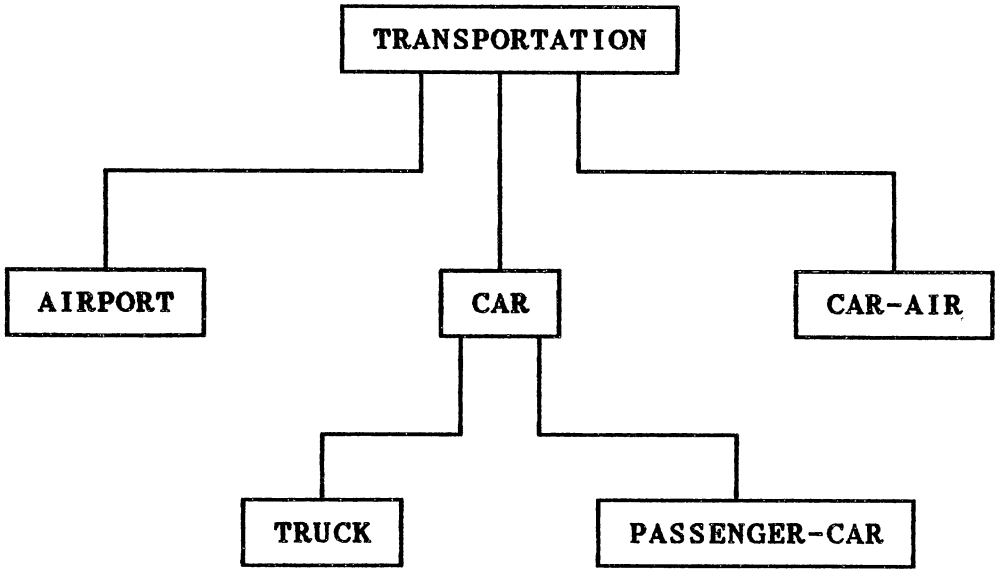


Figure 4-5. A Hierarchical Relationship among ESS Objects in Travel Assistant Expert System.

Expert System defines a tree structure (Figure 4-5). As can be seen in Figure 4-5, an ESS object can not have more than one ESS object parent. In the hierarchical relationship, children ESS objects have the privilege of accessing the Local Working Memory of a parent ESS object. When children ESS objects search for some value in the Local Working Memory of their parent ESS object, the parent ESS object tries to infer the value if it is unknown. An ESS object searches the Local Working Memory of its parent when the variables it needs to evaluate are not declared in its Local Working Memory. This process is called "Local Working Memory Inheritance." In Personal Consultant Plus, parameter groups (i.e., local working

memories) in parent frames can be accessed (i.e., inherited) by children frames. However, the inheritance mechanism does not apply to rule groups (i.e., Rule Bases) [TEXA87]. ESS objects are reusable software components, and when ESS objects are reused in different expert systems, the Local Working Memory Inheritance among ESS objects should be maintained in order to avoid inconsistencies and potential faults.

The Local Working Memory Instance Variable

The Local Working Memory (LWM), which is local to an ESS object, serves as storage for values input by the user and the data inferred by the inference engine object while OOMESS executes a consultation. The Local Working Memory is an aggregate of local working memory elements. The properties of each local working memory element are PARAMETER_NAME, DATA_TYPE, ASK_USER, LEGALVALS, QUESTION, VALUE, and STATIC_OR_DYNAMIC. An example of the properties of a local working memory element is shown in Figure 4-6. Some ideas are adopted from the parameter properties of Personal Consultant Plus in defining the properties of a local working memory element.

The PARAMETER_NAME property specifies the name of a LWM element. The DATA_TYPE is used for declaring the data type of the LWM element. In OOMESS, data types are limited to INTEGER, FLOAT, BOOLEAN, and STRING. The INTEGER type is a signed integer, while the FLOAT type is used for

```
PARAMETER_NAME: DAYTYPE
DATA_TYPE: STRING
ASK_USER: YES
  LEGALVALS: WEEKEND, WEEKDAY
  QUESTION: IS IT WEEKEND OR WEEKDAY?
VALUE: NIL
STATIC_OR_DYNAMIC: STATIC
```

Figure 4-6. The Properties of a Local Working Memory Element.

storing floating point numbers; the ranges in the possible values of these data types depend on the computer system. The BOOLEAN type can have either TRUE or FALSE. The STRING type is used for storing a fixed length character string.

If the value of a LWM element needs to be supplied by the user, the ASK_USER will be set to YES; otherwise, it will be set to NO. The LEGALVALS is used for defining possible values of a LWM element, and the QUESTION is used for specifying a prompt. When the user is prompted for the value of a LWM element, the prompt, included in the LWM element, will appear on the screen. If a LWM element need not consult the user to get a value, the LEGALVALS and the QUESTION will be set to NIL. The VALUE property is initialized to NIL when a LWM element is created. NIL

means that the value of a LWM element is unknown. The VALUE property will change to the value as soon as the value of its LWM element is inferred or input by the user.

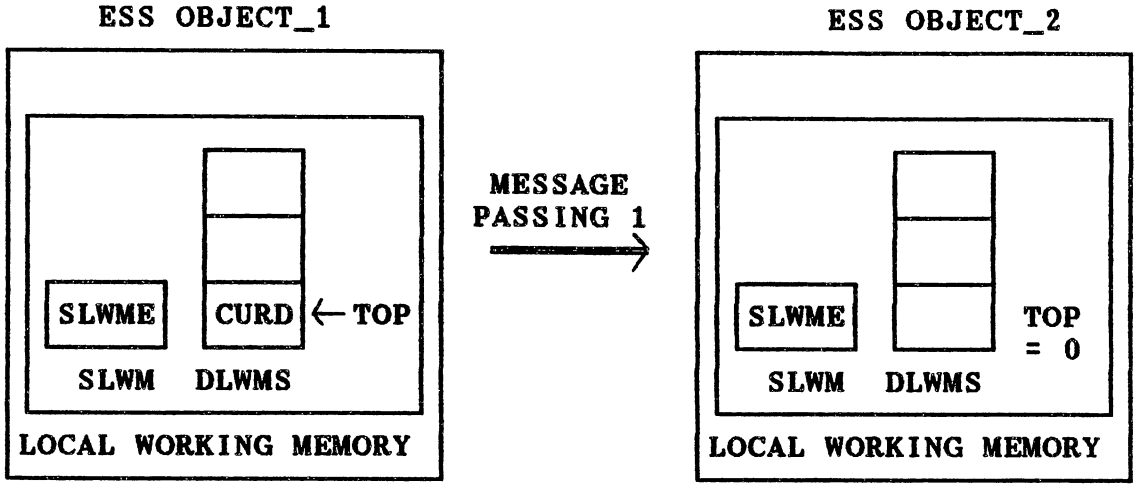
The STATIC_OR_DYNAMIC has a value either STATIC or DYNAMIC. When a LWM element has STATIC as the value of STATIC_OR_DYNAMIC, the LWM element is called a static parameter. The value of a static parameter is persistent; its lifetime extends beyond the duration of the consultation period of the ESS object to the consultation session. A static parameter is similar to a static variable in the C programming language [WAIT87]. Usually, most of the static parameters are templates for user input, and often the user input values are the same whenever an ESS object is activated. If a LWM element has DYNAMIC as the value of STATIC_OR_DYNAMIC, the LWM element (called a dynamic parameter) will be initialized every time its ESS object is activated. That is, the dynamic parameter values are local to the activation of the ESS object, whereas the static parameter values are local to the ESS object.

Local Working Memory Management for Recursive Calls and a Hierarchical Relationship among ESS Objects

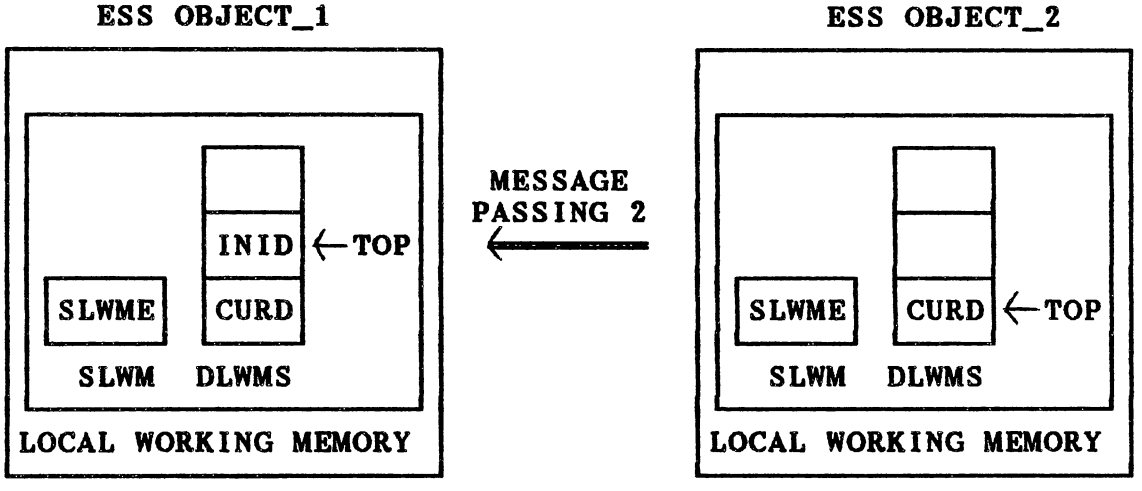
Normally, when a message-receiving ESS object completes its inference and returns control to the message-sending ESS object, only the values of the static parameters of the message-receiving ESS object are retained

and all the values of its dynamic parameters are initialized to NIL. In some cases, however, a message-receiving ESS object sends a message back to a message-sending ESS object to get some value before it infers the final goal value. Since the message-sending object already has some inferred data in its Local Working Memory, the Local Working Memory has to be initialized. Otherwise, already inferred data in the Local Working Memory of the message-sending object affect the result of inference. Furthermore, the initialized Local Working Memory loses old values.

The recursive activations of ESS objects can be solved by the use of a Dynamic Local Working Memory stack. Figures 4-7.1 and 4-7.2 show an example of dealing with a Dynamic Local Working Memory stack. TOP is a pointer to the topmost element of the Dynamic Local Working Memory stack. Local Working Memory is divided into Static Local Working Memory and a Dynamic Local Working Memory stack. Each element of the Dynamic Local Working Memory stack of an ESS object is used for storing only dynamic parameters. Static parameters are stored separately into Static Local Working Memory. When one ESS object performs its consultation, the dynamic parameters pointed by TOP and the static parameters are used in its local working memory. When one ESS object starts inference, its dynamic parameters, whose values are all NIL, are pushed into its Dynamic Local Working Memory stack, and its static



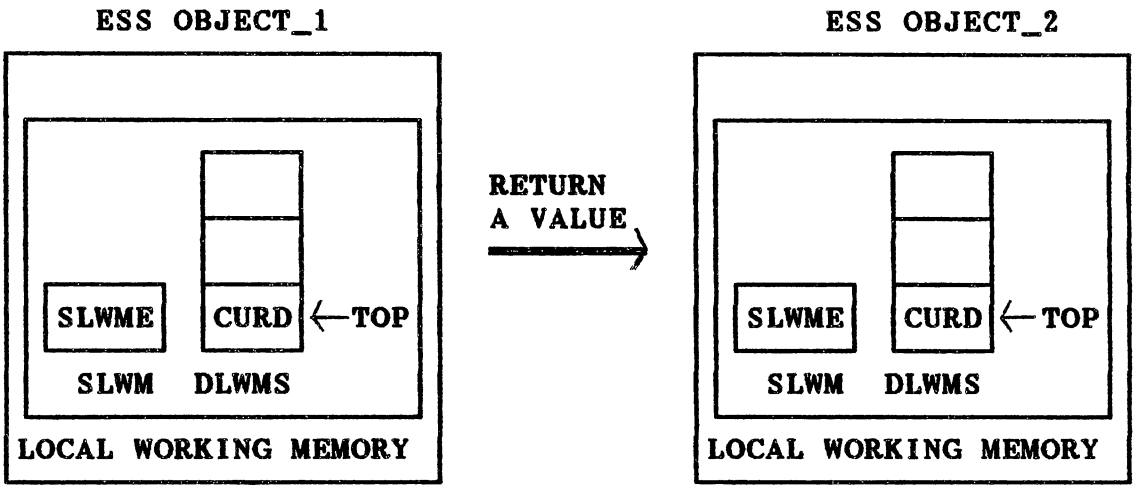
(a) When OBJECT_1 Sends a Message to OBJECT_2 before OBJECT_1 Finishes its Consultation.



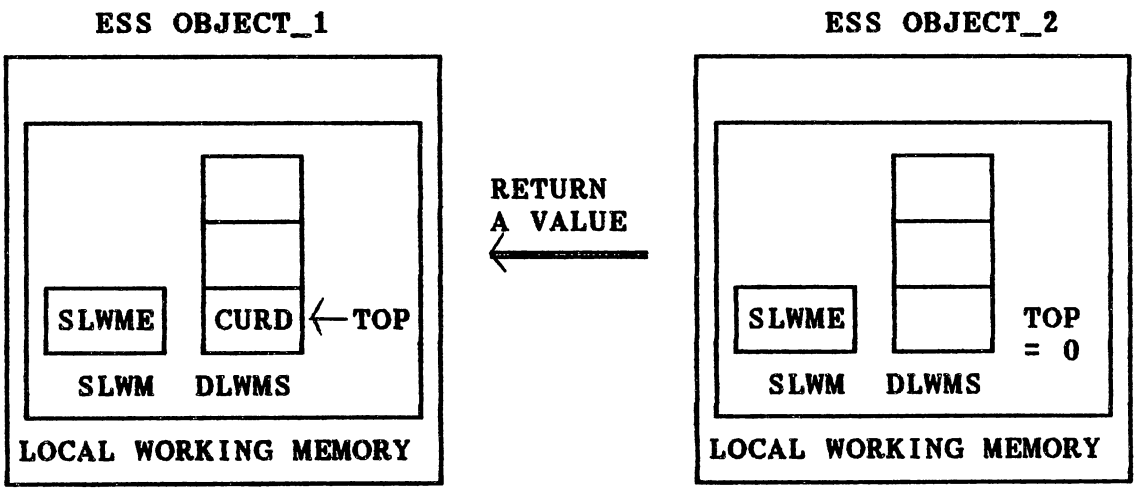
(b) When OBJECT_2 Sends a Message Back to OBJECT_1 before OBJECT_2 Finishes its Consultation.

SLWM: Static Local Working Memory
SLWME: Static Local Working Memory Elements
DLWMS: Dynamic Local Working Memory Stack
INID: Initial Dynamic parameters
CURD: Current inferred Dynamic parameters

Figure 4-7.1. The Way of Dealing with Dynamic Local Working Memory Stack.



(C) When OBJECT_1 Returns a Value to OBJECT_2 as the Responding Answer Value of Message Passing 2.



(d) When OBJECT_2 Returns a Value to OBJECT_1 as the Responding Answer Value of Message Passing 1.

SLWM: Static Local Working Memory
SLWME: Static Local Working Memory Elements
DLWMS: Dynamic Local Working Memory Stack
INID: Initial Dynamic parameters
CURD: Current inferred Dynamic parameters

Figure 4-7.2. The Way of Dealing with Dynamic Local Working Memory Stack.

parameters are stored into its Static Local Working Memory. Therefore, TOP is going to be 1 when an ESS object starts to do inference.

In Figure 4-7.1 (a), OBJECT_1 sends a message to object_2 during its consultation, and OBJECT_2 starts to do its consultation. And then OBJECT_2 sends a message back to OBJECT_1 during its consultation. Because OBJECT_1 does not finish its consultation yet (TOP = 1), initialized dynamic local working memory elements, whose values are all NIL, are pushed into its Dynamic Local Working Memory stack (Figure 4-7.1 (b)). When OBJECT_1 returns a value to OBJECT_2 as the responding answer value of OBJECT_2's message, the Dynamic Local Working Memory stack in OBJECT_1 is popped up (Figure 4-7.2 (c)). Finally, OBJECT_2 returns a value to OBJECT_1 as the response to OBJECT_1's message, and OBJECT_1 continues its consultation. Because OBJECT_2 completely finished its consultation, the TOP of OBJECT_2 has been set to 0 (Figure 4-7.2 (d)).

OOMESS is also designed to provide a hierarchical relationship among ESS objects, from which children ESS objects have the privilege of accessing the Local Working Memory of a parent ESS object. When an ESS object looks for some values in the Local Working Memory of its parent ESS object, there are 3 cases to consider:

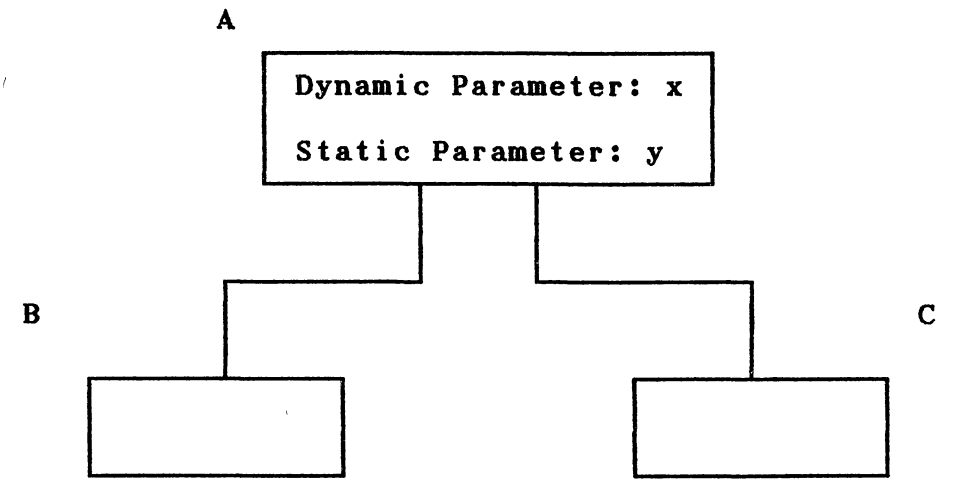
1. The parameter is not declared in its parent's Local Working Memory.
2. The parameter is declared in its parent's Local

Working Memory and the parameter value is known.

3. The parameter is declared in its parent's Local Working Memory and the parameter value is unknown.

In Case 3, the initialized dynamic parameters of the parent object should be pushed into its Dynamic Local Working Memory stack since the parent ESS object tries to infer the parameter value (it is possible that the activation is recursive). In cases 1 and 2, however, the initialized dynamic parameters of the parent object do not need to be pushed into the stack because the parent ESS object does not try to infer the parameter value. In case 1, the parameter value is searched for in the parent Local Working Memory of the parent object, which in turn will repeat the process. Therefore, in Case 3, when the parent ESS object returns to the child object, the parent object pops up its Dynamic Local Working Memory. In cases 1 and 2, however, the parent ESS object should not pop up its Dynamic Local Working Memory.

As an example, let us consider the recursive calls shown in Figure 4-8. ESS object A has a dynamic parameter (x) and a static parameter (y). Assume that the initial values of x and y are 1 and 2 respectively. We further assume that x and y are not declared in ESS object B, but they are used in it. When A calls B (A => B), B can get the values of x and y (x = 1 and y = 2). Let's assume that A gets a new x value (x = 2) when B recursively calls A (A => B => A). At this point, if A recursively calls B (A =>



A => B => A => B

=> means message-sending.

Figure 4-8. An Example of Recursive Calls.

B => A => B), B can get the values of x and y (x = 2 and y = 2): the x value is changed, but the y value is not changed. In order to enforce inheritance, the most recent effective activation of the object is used.

The Rule Base Instance Variable

Domain knowledge is stored in the Rule Bases of ESS objects and user-defined objects. OOMESS supports modularity; therefore a problem can be divided into small subproblems and the knowledge to solve subproblems can be stored in ESS objects. For the division of the main problem into smaller problems, an object-oriented development concept [BOOC86] is used since this concept, as Booch observes, offers a mechanism that captures a model of

the real world.

The Rule Base stores rules that are structured in the form of production rules since production rule representation offers the advantages of understandability and ease of modification [LAUR87]. Production rules in OOMESS have the form "IF antecedent THEN consequent." The antecedent is made up of one or more conditions that are to be matched against Local Working Memory, Global Working Memory, ESS objects, user-defined objects, built-in objects, local rules in the Rule Base, superclass Local Working Memory, and/or input by the user. The consequent specifies the actions to be taken when all the conditions of the antecedent turn out to be true. For making condition and conclusion statements in the antecedent and consequent, parameters, constant values (quoted by " "), and send() functions are used together with other built-in operators (defined in Appendix A). The definitions of operators mostly follow those in the C programming language [WAIT87]. Every parameter should be declared before it is used in rules. A sample Rule Base of an ESS object is shown in Figure 4-9. All rules in the Rule Base are linearly linked from rule1 to rule9. The sample Rule Base, which is the Rule Base of the TRANSPORTATION ESS object, is a portion of the Travel Assistant Expert System Rule Base (Appendix B). In rule1, rule2, and rule3, the send() function is used in antecedent. The send() function is used for sending a message from a rule in Rule Base to a

```
rule1: IF send(airport.cost(Distance)) <= MoneyAvail
       THEN have_money_to_fly.

rule2: IF send(car.cost(Distance)) <= MoneyAvail
       THEN have_money_to_drive.

rule3: IF send(car-air.cost) <= MoneyAvail
       THEN have_money_to_drive_fly.

rule4: IF (Distance / send(airport.mph(Distance))) <=
           TimeAvail
       THEN have_time_to_fly.

rule5: IF (Distance / send(car.mph)) <= TimeAvail
       THEN have_time_to_drive.

rule6: IF send(car-air.have_time)
       THEN have_time_to_drive_fly.

rule7: IF have_money_to_drive AND
          have_time_to_drive
       THEN TravelMode = 'drive'.

rule8: IF have_money_to_drive_fly AND
          have_time_to_drive_fly  AND
       THEN TravelMode = 'drive_and_fly'.

rule9: IF have_money_to_fly   AND
          have_time_to_fly    AND
       THEN TravelMode = 'fly'.
```

Figure 4-9. Sample Rule Base Which is the Rule Base of TRANSPORTATION ESS Object.

```
send( object-name.message-name[(arg1, ..., argN)]
                    [, return-parameter])
```

Data types of arg1, ...,argN, and return-parameter:
STRING, INTEGER, FLOAT, BOOLEAN.
N is an integer number.
[] specifies optional arguments.

Figure 4-10. The Syntax of the Send() Function

specific ESS object, a user-defined object, or a built-in object; therefore, the send() function provides interactions among these objects. The syntax of the send() function is shown in Figure 4-10. The syntax of the first argument in the send() function follows that used for message sending in the C++ programming language [STRO86, COX86].

The first argument of the send() function specifies an object name, a message name, and a passing parameter list. The second argument specifies the returning parameter name in which a resulting value is stored. Therefore, only one value can be received as a result of a message.

For example, in the first argument of the send() function in rule1, "airport" is an ESS object name, "cost" is a message name (i.e., a goal name), and "Distance" is an actual parameter. Therefore, the send() function will send a message to the airport ESS object with the actual parameter "Distance" to infer the value of cost; actually,

cost is not a method in the airport object, but is a goal name in the PASSED_ARGUMENT list of the ESS object airport. The airport object, the message-receiving object, uses one of its PASSED_ARGUMENT lists, which contains cost in the first parameter and one more passing parameter (i.e., (COST, DISTANCE) in Figure 4-4). The value of the actual parameter "Distance" is passed to "DISTANCE." After inferring the value of "cost," the airport object returns that value to the message-sending object (i.e., TRANSPORTATION object) as the reply to the message. If the airport object can't infer the value of cost, the object returns NIL and the inference engine object operating on the TRANSPORTATION object does not consider condition parts of rule1 further, and the antecedent of rule1 becomes false (NIL means unknown or FALSE). If the return parameter is specified in the send() function, the return value will be stored in the local working memory element corresponding to the return-parameter. Because the return-parameter is not specified in the send() function in rule1, a return value will be used only for logical operation with MoneyAvail, and then it will be discarded.

When the send() function sends a message to a user-defined object or a built-in object, in the first argument in the send() function, the object-name argument is the name of a user-defined or a built-in object, the message-name is a method name (i.e., a kind of procedure name) which is inside the message-receiving object, and (arg1,

...,argN) are passing parameters. This send() function will be translated by the system to actual message-sending code, which is language dependent. A user-defined or built-in object can return only one value as the response to the send() function. Since OOMESS keeps names of all ESS objects in a specific expert system, the shell can distinguish ESS objects from both user-defined and built-in objects.

The Inference-Engine-Type Instance Variable

As mentioned earlier, every ESS object can select its own inference strategy using an Inference-Engine-Type instance variable. Therefore, not all ESS objects need to have the same inference strategy. However, for the sake of simplicity, OOMESS currently provides an Inference-Engine class that has only a backward-chaining inference engine object as its instance. An Inference-Engine class can be improved to have several different inference engine objects as its instances (e.g., backward-chaining, forward-chaining, hybrid backward- and forward-chaining inference engine object, etc.). Because the backward-chaining inference engine tries to solve a single goal at any given time, this property provides expert system builders with an easy way of dividing a large rule base into smaller ones. The basic concept of backward-chaining strategy is described in Chapter II.

The backward-chaining inference engine object

operating an ESS object begins by attempting to infer the value of a goal parameter by searching for a rule whose THEN consequent assigns a value to the goal parameter. When it finds a rule, it determines whether the conditions expressed in the rule's "IF" antecedent are true.

To determine whether the conditions expressed in the rule's IF antecedent are true, the inference engine object may need to find the value of one or more parameters included in the IF antecedent, and the inference engine object may need to find the value of one or more parameters included in the right-hand side of the assignment operator (=) in the THEN consequent to assign a value to the parameter in the left-hand side. The method of tracing the value of a parameter included in the IF antecedent or THEN consequent is described below. If the inference engine object, operating on an ESS object, infers the value of the parameter in one of the following steps, it stops the search.

1. If the inference engine object encounters a send construct in which the return-parameter is not specified, one of the following occurs:

   1-1. a message to a specific ESS object. The inference engine object of the message-receiving ESS object will get control of OOMESS. The result of the inference is returned as the response to the message.

   1-2. a message to a user-defined or a built-in object.

The message-receiving object returns the result. The result is not stored in the receiving object Local Working Memory. If the result is NIL, the rule fails. Stop the search.

2. If a parameter is a local parameter (i.e., the parameter is declared in the Local Working Memory),

2-1. search the value of the parameter in the Local Working Memory:

a. if the value of the parameter is known, get the value and stop the search.

b. if the value of the parameter is unknown and the ASK_USER property of the parameter is YES, prompt the user, store the value in the Local Working Memory, and stop the search.

c. if the value of the parameter is unknown and the ASK_USER property of the parameter is NO, go to step 2-2.

2-2. if the inference engine encounters a send construct in which the return-parameter is specified, perform the same process as in step 1 except that the return value is stored in the receiving object Local Working Memory.

2-3. attempt to set the value of the parameter by looking for a rule whose THEN consequent assigns a value to the parameter. When it finds a rule, it determines whether the conditions expressed in the rule's IF antecedent are true. If the conditions

are true, the THEN consequent assigns the value to the parameter and the parameter value is stored in Local Working Memory; otherwise, look for another rule whose THEN consequent assigns a value to the parameter.

3. If a parameter is not a local parameter (i.e., the parameter is declared in Global Working Memory or the parent local working memories),

3-1. search the value of the parameter in the parent Local Working Memory:

a. if the parameter is not declared in the parent Local Working Memory, search the parameter value in the parent Local Working Memory of the parent object, and so forth.

b. if the parameter is declared in the parent Local Working Memory and the value of the parameter is known, get the value.

c. if the parameter is declared in the parent Local Working Memory and the value of the parameter is unknown, the parent object starts to infer the value of the parameter. If the parameter is declared in the GOAL_CANDIDATES of the parent object, the inference engine object of the parent object tries to solve that goal value; otherwise, the inference engine object tries to solve all goals in the GOAL_CANDIDATES until the parameter value is found.

Finally, if the parameter value is unknown, go to step 3-2; otherwise, stop the search.

3-2. search the parameter value in Global Working Memory. Perform the same process as in step 2-1 except that the searching takes place in Global Working Memory. If the parameter value is not found, go to step 3-3.

3-3. if the inference engine encounters a send construct in which the return-parameter is specified, perform the same process as in step 1 except that the return value is stored in the receiving object Global Working Memory.

3-4. the same process as in step 2-3 except that the inferred parameter value is stored in Global Working Memory.

The inference engine object continues the above searches for parameter values when it evaluates the antecedent of a rule, up to the point when it determines that the antecedent will pass or fail. The rule "IF (P_1 AND P_2 AND ... P_n) THEN S" fails if any condition in a series connected by ANDs fails, and the rule "IF (P_1 OR P_2 OR ... P_n) THEN S" fails if a series of conditions connected by ORs do not contain at least one passing condition.

As soon as the inference engine object determines that a rule will pass or fail, it stops tracing parameter values in that rule. It does not attempt to trace any antecedent

parameters that have not yet been evaluated. As a result, the inference engine object does not prompt the user for unneeded information or perform inference logic that does not contribute to the result.

As an example, let's consider the rule base of the TRANSPORTATION ESS object (Figure 4-9 and APPENDIX B). Let us assume that TravelMode is the goal. The consequents of rule7, rule8, and rule9 assign values to the goal parameter TravelMode. The inference engine object operating on the TRANSPORTATION object starts from rule7 to determine whether the conditions expressed in rule7's "IF" antecedent are true. The parameter have_money_to_drive, which is the first condition in rule7's "IF" antecedent, is declared in the Local Working Memory of TRANSPORTATION object, the ASK_USER property of the parameter is NO, and the value of the parameter is unknown (step 2-1.c). Therefore, rule2 is applied to get the value of have_money_to_drive (step 2-3). To determine whether the condition expressed in rule2's "IF" antecedent is true, the expression "send(car.cost(Distance))" should be evaluated first. The send() function sends a message to the ESS object car with the actual parameter "Distance" to infer the value of cost (step 1-1). Because the value of Distance is unknown, the user is prompted for the value before the send() function sends a message (step 2-1.b): Distance is declared as a local parameter and the ASK_USER property of the parameter is YES. For receiving passing parameters, the car object,

the message-receiving object, uses the list (COST,DISTANCE) which is one of its PASSED_ARGUMENT lists. After inferring the value of "cost," the car object returns the value of "cost" as the reply to the TRANSPORTATION object. Then, the user is prompted for the value of MoneyAvail (step 3-2): MoneyAvail is declared as a global parameter and the ASK_USER property of the parameter is YES. After getting the value of MoneyAvail, the logical operation "<=" is performed. Because a return-parameter is not specified in the send() function in rule2, the return value is used only for logical operation with MoneyAvail, and then it is discarded. If the result of the logical operation is true, have_money_to_drive gets the value TRUE, and the second condition in rule7 is considered. If two conditions in rule7 turn out to be true, the goal TravelMode gets the value 'drive' and the consultation concludes.

User Interface

The role of the user interface in OOMESS is to provide the tools for the user to create expert systems (i.e., ESS objects, Global Working Memory, and user-defined objects), and to perform consultations [HEND88]. Further, the user interface may provide a rule network browser, which lets the user see every logical link between rules [NEUR87], and a Local Working Memory dependency network, which shows Local Working Memory dependencies among local working memories.

For example, when the user creates the "Travel Assistant Expert System," the user interface prompts the user for the type of knowledge he wants to create: ESS objects, Global Working Memory, and user-defined objects. To create ESS objects in the Travel Assistant Expert System (i.e., TRANSPORTATION, AIRPORT, CAR, CAR-AIR, TRUCK, and PASSENGER-CAR), the user types in ESS objects. That action makes the user interface call the "New" method, defined in the ESS class, which helps the user to create a new ESS object with its attribute values for Inference-Engine-Type, Object-Parameters, Rule Base, and Local Working Memory. After creating all ESS objects, the user types in Global Working Memory to create Global Working Memory elements (i.e., MoneyAvail), which are stored into Global Working Memory. If the user types in user-defined objects, the user interface provides the editor to help the user create user-defined objects in the host language (i.e., the language which provides the environment for OOMESS). The user interface also helps the user to perform consultations; this feature will be described in the following section.

Consultation

When the consultation begins, the user types in a specific expert system name. Then, OOMESS loads all ESS and user-defined objects that are part of the expert system, and lists goal parameter names under the name of

every ESS object on the screen. OOMESS obtains the goal parameter names from the GOAL_CANDIDATES property which is defined as part of the Object-Parameters of every ESS object. The user selects a specific goal name in a specific ESS object, which means that every ESS object can be directly selected by the user when the consultation starts. OOMESS now starts to infer the goal value with the selected ESS object. Normally, OOMESS infers the goal value through message-passing among the ESS objects of a specific expert system. Some values can be prompted to the user except for the goal value if it is necessary during the consultation; this means that the inference engine object in every ESS object uses information from both the knowledge base and the user in order to arrive at a conclusion.

As an example, let's consider a sample consultation of the Travel Assistant Expert System (APPENDIX B). To begin the consultation, the user types in "Travel Assistant Expert System". Then, OOMESS loads all ESS objects that are part of the Travel Assistant Expert System (i.e., TRANSPORTATION, AIRPORT, CAR, CAR-AIR, TRUCK, and PASSENGER-CAR). And then, OOMESS lists goal parameter names under the name of every ESS object on the screen. We assume that the user selects the goal "mph" in the CAR object; the GOAL_CANDIDATES of the CAR object contains cost, $_per_mile_drive, and mph (APPENDIX B). The CAR ESS object now starts to infer the goal value. The consequents

of rule5 and rule6 assign values to the goal parameter mph (APPENDIX B). The inference engine object operating on the CAR object starts from rule5 to determine whether the condition expressed in the antecedent of rule5 is true. The parameter Car_Type, which appears in the antecedent of rule5, is declared in the Local Working Memory of the CAR object; the ASK_USER property of the parameter is YES; the value of the parameter is unknown. Therefore, the user is prompted for the value of Car_Type. After getting the value, it is compared with 'Truck'. If the result of the logical comparison is true, the return value of the send construct "send(Truck.mph)", which appears in rule5's consequent, is assigned to mph. When the value of mph is found, the consultation ends.

CHAPTER V

SUMMARY, CONCLUSION, AND SUGGESTED FUTURE WORK

OOMESS (an Object-Oriented Modular Expert System Shell) is an expert system shell which is designed with the objective of integrating a production system with an object-oriented language. This thesis describes the design of OOMESS. The view projected by OOMESS consists of a collection of objects with the capability of communicating with each other. OOMESS provides a backward-chaining inference strategy, including access from rules to the functionality of an object-oriented language (i.e., built-in objects), rule group objects (i.e., ESS objects), and user-defined objects.

The OOMESS approach to problem solving encourages a problem to be divided into subproblems according to the level of detail of the problem. Each subproblem has a knowledge base associated with it. These separated knowledge bases are stored in ESS objects and/or user-defined objects. ESS objects and/or user-defined and built-in objects are selected during the system execution, and the goal-directed nature of the system guides the search through these selected objects.

There are many advantages to OOMESS:

1. It provides an efficient mechanism for managing a large rule set compared with expert system shells that do not provide rule groups. In OOMESS, a large rule base is modularized into smaller rule bases and user-defined objects. This feature should provide an efficient mechanism for managing a large rule base.
2. It provides very flexible interactions among rule group objects (i.e., ESS objects) compared with current commercial expert system shells since one rule group object can send a message to any of the other rule group objects, and a message-receiver is allowed to recursively send a message to the message-sender.
3. It eases understanding of complex systems because the object-oriented concept can be used to design expert systems that consist of interacting modules.
4. It provides an incremental method of developing expert systems because a large knowledge base is divided into smaller knowledge bases and stored into rule group objects and user-defined objects.
5. It provides access from rules to rule group objects, user-defined objects, and built-in objects.
6. It allows the user to select one of the rule group objects in a specific expert system when the consultation starts. Therefore, any rule group object can be used to trigger inference at the beginning of the consultation. Personal Consultant Plus, by

comparison, always starts the consultation from a root rule group object (i.e., root frame).

7. It supports reusability. ESS objects are reusable software components, and when they are reused in different expert systems, the Local Working Memory Inheritance among them should be maintained.

OOMESS, however, does not possess all the properties of an object-oriented system because it does not incorporate class hierarchy among ESS objects (i.e., every ESS object is an instance of the ESS class). But OOMESS is designed to provide a hierarchical relationship among the ESS objects and Local Working Memory Inheritance. The children ESS objects have the privilege of having access to the Local Working Memory of a parent ESS object. If the values of local working memory elements in a parent ESS object are unknown when children ESS objects search for some values in the Local Working Memory of the parent ESS object, the parent ESS object tries to infer the values. User-defined and built-in objects, however, have both class hierarchy and inheritance properties as provided by their language of definition.

The following improvements are suggested for future work:

1. In the present design, only one inference strategy is used. However, it is possible to provide several different inference strategies. Since every ESS object can select its own inference strategy using the

Inference-Engine-Type instance variable, all ESS objects do not need to have the same inference strategy. OOMESS, however, currently provides an Inference-Engine class that has only a backward-chaining inference engine object as its instance.

2. The capability to dynamically allocate and deallocate ESS objects at execution time needs to be added.
3. Explanation facilities [MART88] which inform the user of the reasoning path OOMESS is taking to solve the specific problem need to be provided.
4. Certainty factor [MART88, PARS88] which is one method used for dealing with uncertainties in rule base systems can be added to OOMESS.
5. Class hierarchy and inheritance properties among rule group objects can be also added to OOMESS.

In the design of OOMESS, we have attempted to make provisions for these features to be added in future work. OOMESS is a new object-oriented expert system shell. It can evolve and mature in future work.

BIBLIOGRAPHY

[ALLE83] Allen, L., "YAPS: Yet Another Production System," Technical Report TR-1146, Department of Computer Science, University of Maryland, (1983).

[ARCI88] Arcidiacono, T., "Expert System On-call," PC TECH JOURNAL, (Nov. 1988), 112-114.

[BOBR83] Bobrow, D. G., and Stefik, M., "The LOOPS Manual," Xerox Palo Alto Research Center, (Dec. 1983).

[BOOC86] Booch, G., "Object-Oriented Development," IEEE Software, vol. SE-12, No 2 (Feb. 1986), 211-221.

[BROW88] Brown, C., and Subramanian, S., "SOFTWARE REVIEWS: Powerful, visual expert-system shell," IEEE Software, (Sept. 1988), 98-100.

[BUCH84] Buchanan, B. G., and Shortliffe, E. H. (eds.), "Rule-based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project," Addison-Wesley, Reading, Mass., (1984).

[CASA88] Casais, E., "An object oriented system implementing KNOs," In Proceeding Conference on Office Information Systems. (Palo Alto, CA, March 23-35). ACM, New York, (1988), 284-290.

[COX84] Cox, B. J., "Message/Object programming: An Evolutionary Change in Programming Technology," IEEE Software, vol. 1(1) (Jan. 1984), 50-61.

[COX86] Cox, B. J., "Object-Oriented Programming," Addison-Wesley, Reading, Mass., (1986).

[DAHL66] Dahl, O. J., and Nygaard, K., "SIMULA -a goal-based simulation language," Communications of the ACM. 9, (1966), 671-678.

[DAVI77] Davis, R., and King, J., "An overview of production systems," In Machine Intelligence 8: Machine Representations of Knowledge, E. Elcock, and D. Michie, Eds, Wiley, New York, (1977), 300-332.

[FIKE85] Fikes, R., and Kehler, T., "The Role of frame-based representation in reasoning," CACM 28, 9 (Sept. 1985), 904-920.

[FIRE88] Firebaugh, M. W., "Artificial Intelligence: a knowledge-based approach," Boyd & Fraser, Boston. (1988).

[GOLD83] Goldberg, A., and Robson, D., "Smalltalk-80: The language and its implementation," Addison-Wesley, Reading, Mass., (1983).

[GUTM89] Gutman, A., "Object-Oriented Programming in AI: New Choices," AI Expert, (Jan. 1989), 53-71.

[HEND88] Hendler, J. A., "EXPERT SYSTEMS: THE USER INTERFACE," Ablex Publishing Corporation, Norwood, New Jersey 07648, (1988).

[KEHL84] Kehler, T. P., and Clemenson, G. D., "An application development system for expert systems," Syst. Softw. 3,1 (Jan., 1984), 212-224.

[LAUR87] Laursen, J., and Atkinson, R., "Opus: A Smalltalk Production System," OOPSLA '87 Proceedings, (Oct. 4-8, 1987), 377-387.

[LISK75] Liskov, B. H., and Zilles, S. N., " Specification Techniques for Data Abstractions," IEEE Software, vol. SE-1, No. 1 (March 1975), 7-19.

[MART88] Martin, J., and Oxman, S., "Building Expert Systems," Prentice-Hall, Englewood Cliffs, N.J., (1988).

[MINS75] Minsky, M., "A framework for representing knowledge," In The Psychology of Computer Vision, P. Winston, Ed. McGraw-Hill, New York, (1975), 211-277.

[NEUR87] Neuron Data., "Nexpert Object," 444 High Street, Palo Alto, CA 94301, (1987).

[PARN72] Parnas, D. L., "On the criteria to be used in decomposing systems into modules," Commun. ACM, (Dec. 1972), 1053-1058.

[PARS88] Parsaye, K., and Chignell, M., "EXPERT SYSTEMS FOR EXPERTS," John Wiley & Sons, Inc, New York, (1988).

[PIER86] Piersol, K. W., "The HUMBLE Reference Manual," Xerox Special Information Systems, (1986).

[RENT82] Rentsch, T., "Object-Oriented programming," SIGPLAN Notices, vol. 17, no. 9 (Sept. 1982), 51-57.

[SAUE83] Sauers, R., and Walsh, R., "On the requirements of future expert systems," IJCAI-83, 1, (1983), 110-115.

[SIEG86] Siegel, P., "EXPERT SYSTEMS: A NON-PROGRAMMER'S GUIDE TO DEVELOPMENT AND APPLICATIONS," TAB BOOKS Inc., P.O. Box 40, Blue Ridge Summit, PA 17214, (1986).

[STEF86] Stefik, M., and Bobrow, D. G., "Object-Oriented Programming: Themes and Variations," AI magazine 6, 4 (Winter 1986), 40-62.

[STRO86] Stroustrup, Bjarne., "The C++ Programming Language," Addison-Wesley, Reading, Mass., (1986).

[TEXA87] Texas Instruments Incorporated., "Personal Consultant Plus Reference Guide," Data Systems Group, Austin, Texas. (Aug. 1987).

[TOKO85] Tokoro, M., and Ishikawa, Y., "Oriented84/K: A Language Within Multiple Paradigms in the Object Framework," Dept of Electrical Engineering, Keio University, 3-14-1 Hiyoshi, Yokohama 223, Japan, (1985).

[WAIT87] Waite, M., and Prata, S., and Martin, D., "The Waite Group C Primer Plus User-Friendly Guide to the C Programming Language," HOWARD W. SAMS & COMPANY, Indianapolis, Indiana, (1987).

APPENDIX A

OPERATORS THAT ARE USED IN ANTECEDENT AND CONSEQUENT IN PRODUCTION RULES

Operator in Order of Increasing Precedence:

=	Associativity is right to left.
+ -	Associativity is left to right.
* /	Associativity is left to right.
( )	Associativity is left to right.

Assignment Operator:

=	Assigns its right value to the its left variable.

Arithmetic Operator:

+	Adds its right value to its left value.
-	Subtracts its right value to its left value.
*	Multiplies its right value by its left value.
/	Divides its left value by its left value.

Relational Operators:

Each of these operators is used for comparing its left value to its right value.

<	less than

<=	less than or equal to
==	equal to
>=	greater than or equal to
>	greater than
!=	unequal to

A simple relational expression consists of a relational operator with its left and right operands. If a relational expression becomes true, the relational expression has the value TRUE. If a relational expression becomes false, the relational expression has the value FALSE.

Logical Operators:

Each of these operators, which has relational expressions as operands, is used for logical operation.

AND	logical 'and' operation.
OR	logical 'or' operation.
!	logical 'not' operation.

Logical Expressions:

expression1 AND expression2	is true if and only if both expression1 and expression2 are true.
expression1 OR expression2	is true if both expression1 and expression2 or either one is true.
! expression1	is true if expression1 is

false.

Logical expressions are evaluated from left to right; evaluation terminates as soon as there is evidence that the expression becomes false.

APPENDIX B

TRAVEL ASSISTANT EXPERT SYSTEM

The Travel Assistant Expert System is a sample expert system which can be developed and executed under OOMESS. This expert system helps travelers decide the mode of transportation.

GLOBAL_WORKING MEMORY ELEMENTS

```
PARAMETER_NAME: MoneyAvail
DATA_TYPE: FLOAT
ASK_USER: YES
    LEGALVALES: POSITIVE FLOAT NUMBER ( $ XXXXX.XX )
    QUESTION:  How much money do you have for
                        travel?
VALUE: NIL
```

OBJECT :: TRANSPORTATION

OBJECT_PARAMETERS

```
OBJECT_NAME: TRANSPORTATION
GOAL_CANDIDATES: TravelMode
PASSED_PARAMETER: (TravelMode, Distance), (TravelMode)
SUPER_OBJECT: NIL
```

INFERENCE_ENGINE_TYPE: BACKWARD

LOCAL WORKING MEMORY ELEMENTS

```
PARAMETER_NAME: have_money_to_fly, have_money_to_drive,
                have_money_to_drive_fly,
```

```
                have_time_to_drive, have_time_to_drive_fly,
                have_time_to_fly
DATA_TYPE: BOOLEAN
ASK_USER: NO
    LEGALVALES: NIL
    QUESTION:  NIL
VALUE: NIL
STATIC_OR_DYNAMIC: DYNAMIC

PARAMETER_NAME: TravelMode
DATA_TYPE: STRING
ASK_USER: NO
    LEGALVALES: NIL
    QUESTION:  NIL
VALUE: NIL
STATIC_OR_DYNAMIC: DYNAMIC

PARAMETER_NAME: Distance
DATA_TYPE: FLOAT
ASK_USER: YES
    LEGALVALES: POSITIVE FLOAT NUMBER (UNIT: MPH)
    QUESTION:  How far do you travel?
VALUE: NIL
STATIC_OR_DYNAMIC: DYNAMIC

PARAMETER_NAME: TimeAvail
DATA_TYPE: FLOAT
ASK_USER: YES
    LEGALVALES: POSITIVE FLOAT NUMBER (UNIT: HOUR)
    QUESTION:  How much time do you have for travel?
VALUE: NIL
STATIC_OR_DYNAMIC: STATIC
```

RULE_BASE

```
rule1: IF send(airport.cost(Distance)) <= MoneyAvail
       THEN have_money_to_fly.

rule2: IF send(car.cost(Distance)) <= MoneyAvail
       THEN have_money_to_drive.

rule3: IF send(car-air.cost) <= MoneyAvail
       THEN have_money_to_drive_fly.

rule4: IF (Distance / send(airport.mph(Distance))) <=
           TimeAvail
       THEN have_time_to_fly.

rule5: IF (Distance / send(car.mph)) <= TimeAvail
       THEN have_time_to_drive.
```

```
rule6: IF send(car-air.have_time)
       THEN have_time_to_drive_fly.

rule7: IF have_money_to_drive AND
          have_time_to_drive
       THEN TravelMode = 'drive'.

rule8: IF have_money_to_drive_fly AND
          have_time_to_drive_fly  AND
       THEN TravelMode = 'drive_and_fly'.

rule9: IF have_money_to_fly   AND
          have_time_to_fly    AND
       THEN TravelMode = 'fly'.
```

OBJECT :: AIRPORT

OBJECT PARAMETERS

```
OBJECT_NAME: AIRPORT
GOAL_CANDIDATES: cost, mph, $_per_mile_fly
PASSED_PARAMETER: (cost, Distance), (cost),
                  (mph, Distance), (mph)
SUPER_OBJECT: TRANSPORTATION
```

INFERENCE_ENGINE_TYPE: BACKWARD

LOCAL WORKING MEMORY ELEMENTS

```
PARAMETER_NAME: $_per_mile_fly
DATA_TYPE: FLOAT
ASK_USER: NO
    LEGALVALES: NIL
    QUESTION:  NIL
VALUE: NIL
STATIC_OR_DYNAMIC: DYNAMIC

PARAMETER_NAME: Distance
DATA_TYPE: FLOAT
ASK_USER: YES
    LEGALVALES: POSITIVE FLOAT NUMBER (UNIT: MPH)
    QUESTION:  How far do you travel?
VALUE: NIL
STATIC_OR_DYNAMIC: DYNAMIC

PARAMETER_NAME: cost
DATA_TYPE: FLOAT
ASK_USER: NO
```

```
    LEGALVALES: NIL
    QUESTION:  NIL
VALUE: NIL
STATIC_OR_DYNAMIC: DYNAMIC

PARAMETER_NAME: mph
DATA_TYPE: INTEGER
ASK_USER: NO
    LEGALVALES: NIL
    QUESTION:  NIL
VALUE: NIL
STATIC_OR_DYNAMIC: DYNAMIC
```

RULE BASE

```
rule1: IF Distance >= 1000
       THEN $_per_mile_fly = 0.20.

rule2: IF Distance < 1000 AND
          Distance >= 150
       THEN $_per_mile_fly = 0.60.

rule3: IF Distance < 150
       THEN $_per_mile_fly = 1.00.

rule4: IF TRUE
       THEN cost = $_per_mile_fly * Distance.

rule5: IF Distance <= 150
       THEN mph = 250.

rule6: IF Distance > 150 AND
          Distance < 500
       THEN mph = 350.

rule7: IF Distance >= 500
       THEN mph = 400.
```

OBJECT :: CAR

OBJECT PARAMETERS

```
OBJECT_NAME: CAR
GOAL_CANDIDATES: cost, $_per_mile_drive, mph
PASSED_PARAMETER: (cost, Distance), (cost)
                  ($_per_mile_drive), (mph)
SUPER_OBJECT: TRANSPORTATION
```

INFERENCE_ENGINE_TYPE: BACKWARD

LOCAL WORKING MEMORY ELEMENTS

```
PARAMETER_NAME: cost
DATA_TYPE: FLOAT
ASK_USER: NO
    LEGALVALES: NIL
    QUESTION:  NIL
VALUE: NIL
STATIC_OR_DYNAMIC: DYNAMIC

PARAMETER_NAME: $_per_mile_drive
DATA_TYPE: FLOAT
ASK_USER: NO
    LEGALVALES: NIL
    QUESTION:  NIL
VALUE: NIL
STATIC_OR_DYNAMIC: DYNAMIC

PARAMETER_NAME: Distance
DATA_TYPE: FLOAT
ASK_USER: YES
    LEGALVALES: POSITIVE FLOAT NUMBER (UNIT: MPH)
    QUESTION:  How far do you travel?
VALUE: NIL
STATIC_OR_DYNAMIC: DYNAMIC

PARAMETER_NAME: Num_of_Cylinder
DATA_TYPE: STRING
ASK_USER: YES
    LEGALVALES: 4_Cylinder, 6_Cylinder, 8_Cylinder
    QUESTION:  What kind of car do you have?
VALUE: NIL
STATIC_OR_DYNAMIC: STATIC

PARAMETER_NAME: Road_Type
DATA_TYPE: STRING
ASK_USER: YES
    LEGALVALES: Highway, Local_Road
    QUESTION:  What is road type?
VALUE: NIL
STATIC_OR_DYNAMIC: STATIC

PARAMETER_NAME: Car_Type
DATA_TYPE: STRING
ASK_USER: YES
    LEGALVALES: Truck, Passenger_Car
    QUESTION:  What kind of car do you have?
VALUE: NIL
STATIC_OR_DYNAMIC: STATIC
```

```
PARAMETER_NAME: mph
DATA_TYPE: INTEGER
ASK_USER: NO
    LEGALVALES: NIL
    QUESTION:  NIL
VALUE: NIL
STATIC_OR_DYNAMIC: DYNAMIC
```

RULE BASE

```
rule1: IF Num_of_Cylinder == '4_Cylinder'
       THEN $_per_mile_drive = 0.03

rule2: IF Num_of_Cylinder == '6_Cylinder'
       THEN $_per_mile_drive = 0.05

rule3: IF Num_of_Cylinder == '8_Cylinder'
       THEN $_per_mile_drive = 0.06

rule4: IF TRUE
       THEN cost = $_per_mile_drive * Distance

rule5: IF Car_Type == 'Truck'
       THEN mph = send(Truck.mph).

rule6: IF Car_Type == 'Passenger_Car'
       THEN mph = send(Passenger-Car.mph).
```

OBJECT :: CAR-AIR

OBJECT PARAMETERS

```
OBJECT_NAME: CAR-AIR
GOAL_CANDIDATES: cost, have_time
PASSED_PARAMETER: (cost)
                  (have_time)
SUPER_OBJECT: TRANSPORTATION
```

INFERENCE ENGINE TYPE: BACKWARD

LOCAL WORKING MEMORY ELEMENTS

```
PARAMETER_NAME: Air_Cost
DATA_TYPE: FLOAT
```

```
ASK_USER: NO
    LEGALVALES: NIL
    QUESTION:  NIL
VALUE: NIL
STATIC_OR_DYNAMIC: DYNAMIC

PARAMETER_NAME: Air_Distance
DATA_TYPE: FLOAT
ASK_USER: YES
    LEGALVALES: POSITIVE FLOAT NUMBER (UNIT: MPH)
    QUESTION:  How far do you travel by airplane?
VALUE: NIL
STATIC_OR_DYNAMIC: DYNAMIC

PARAMETER_NAME: Road_Distance
DATA_TYPE: FLOAT
ASK_USER: YES
    LEGALVALES: POSITIVE FLOAT NUMBER (UNIT: MPH)
    QUESTION:  How far do you travel by car?
VALUE: NIL
STATIC_OR_DYNAMIC: DYNAMIC

PARAMETER_NAME: Car_Cost
DATA_TYPE: FLOAT
ASK_USER: NO
    LEGALVALES: NIL
    QUESTION:  NIL
VALUE: NIL
STATIC_OR_DYNAMIC: DYNAMIC

PARAMETER_NAME: have_time
DATA_TYPE: BOOLEAN
ASK_USER: NO
    LEGALVALES: NIL
    QUESTION:  NIL
VALUE: NIL
STATIC_OR_DYNAMIC: DYNAMIC

PARAMETER_NAME: cost
DATA_TYPE: FLOAT
ASK_USER: NO
    LEGALVALES: NIL
    QUESTION:  NIL
VALUE: NIL
STATIC_OR_DYNAMIC: DYNAMIC
```

RULE BASE

```
rule1: IF send(airport.cost(Air_Distance),Air_Cost) AND
          send(car.cost(Road_Distance),Car_Cost)
       THEN cost = Air_Cost + Car_Cost
```

```
rule2: IF ((Air_Distance / send(airport.mph(Air_Distance)))
          + (Road_Distance / send(car.mph)))
          <= TimeAvail
       THEN have_time
```

OBJECT :: TRUCK

OBJECT PARAMETERS

```
OBJECT_NAME: TRUCK
GOAL_CANDIDATES: mph
PASSED_PARAMETER: (mph)
SUPER_OBJECT: CAR
```

INFERENCE_ENGINE_TYPE: BACKWARD

LOCAL WORKING MEMORY ELEMENTS

```
PARAMETER_NAME: mph
DATA_TYPE: INTEGER
ASK_USER: NO
    LEGALVALES: NIL
    QUESTION:  NIL
VALUE: NIL
STATIC_OR_DYNAMIC: DYNAMIC
```

RULE_BASE

```
rule1: IF Road_Type == 'Local_Road'
       THEN mph = 45.

rule2: IF Road_Type == 'Highway'
       THEN mph = 55.
```

OBJECT :: PASSENGER-CAR

OBJECT PARAMETERS

```
OBJECT_NAME: PASSENGER-CAR
GOAL_CANDIDATES: mph
PASSED_PARAMETER: (mph)
SUPER_OBJECT: CAR
```

INFERENCE_ENGINE_TYPE: BACKWARD

LOCAL WORKING MEMORY ELEMENTS

```
PARAMETER_NAME: mph
DATA_TYPE: INTEGER
ASK_USER: NO
    LEGALVALES: NIL
    QUESTION:  NIL
VALUE: NIL
STATIC_OR_DYNAMIC: DYNAMIC
```

RULE BASE

```
rule1: IF Road_Type == 'Local_Road'
       THEN mph = 55.

rule2: IF Road_Type == 'Highway'
       THEN mph = 65.
```

VITA

Jin Cheon Na

Candidate for the Degree of

Master of Science

Thesis: AN OBJECT-ORIENTED MODULAR EXPERT SYSTEM SHELL

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Seoul, Korea, July 30, 1964, the son of Joong Bae Na and Young Keun Na.

Education: Graduated from Young-Il High School, Seoul, Korea, in February, 1983; received Bachelor of Science Degree in Electrical Engineering from Hanyang University in February, 1987; completed the requirements for the Master of Science degree at Oklahoma State University in May, 1990.

Professional Experience: Teaching Assistant, Department of Computing and Information Science, Oklahoma State University, August, 1989, to present.