1988

A virtual operator technique for enhancement of computer-to-computer interactivity

Ying-Chan Fred Wu
Iowa State University

A virtual operator technique for enhancement of computer-to-computer interactivity

Wu, Ying-Chan Fred, Ph.D.

Iowa State University, 1988

A virtual operator technique for enhancement of computer-to-computer interactivity

by

Ying-Chan Fred Wu

A Dissertation Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Department: Electrical Engineering and Computer Engineering
Major: Computer Engineering

Approved:

In Charge of Major Work

For the Major Department

For the Graduate College

Iowa State University
Ames, Iowa

1988

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

When a user sits before a terminal, he or she usually accesses resources as if they were directly connected to the terminal (Figure 1), even though in fact resources the user is dealing with may be located far away and physically connected through multiple levels of transmission (Figure 2). Networking problems illustrated in this latter figure may have been handled by a computer system, making complex procedures transparent to the end user. A problem may arise when a requested resource is in a disjoint system as, for example, when we wish to retrieve a remote WYLBUR file to a Unix system. How do we handle such problems as system adaptivity (Figure 3), to let the user access facilities in other systems?

One solution is to build a remote host program as a terminal emulator, setting up a logical link from the remote system to the user's terminal, letting the user feel as if he or she were in the environment of the connected remote system (Figure 4). The remote host solution can address the problem of reaching a different system, but problems may still remain, since the user may still need to manipulate a system different from one with which he is familiar. For an ideal design, the environmental change should not be reflected to the user, the remote host solution obviously not meeting this requirement.

In order to make an additional shell to deal with the problem of intersystem communication, an interpreter may be used. For each different system connected, such an interpreter may be installed to translate its commands into equivalent commands compatible to the destined system (Figure 5). Using such command translation software is a traditional way to handle the problem of system compatibility. Since the translation and service routines are fixed by the software, it is virtually impossible for a user to add or update the translation routine. Moreover, in the case where a new system is to be added, new translation software will be required.

This suggests the concept of a "Virtual Operator" (Figure 6), an expert system software implementation based on Artificial Intelligence (AI) strategies. The research in this dissertation proposes such a method to enhance interactivity between systems. Achievement of system adaptivity to environmental changes is attained by formulating human-like learning and decision capability in the knowledge base of the virtual operator.

The virtual operator will connect the terminal to the requested system, translate the user's commands to equivalent operations in the connected system, and let a user with only limited computer background communicate with different systems as desired. Additionally, because of the knowledge base contained in the virtual operator, it can be trained by a human expert; experiences during the communication with different systems can be learned and accumulated to improve performance in subsequent communications.

Based on the principles of building an expert system, decision rules (which control the whole learning and communication process), and application program modules are implemented separately, making it relatively easier to change the software in the virtual operator to accept a new system or a change in the command set.

In order to verify the methodology of the virtual operator approach, several systems were chosen for testing, and the virtual operator now has been trained and is capable of communicating with NAS9160/WYLBUR, VAX/VMS and VAX/UNIX.

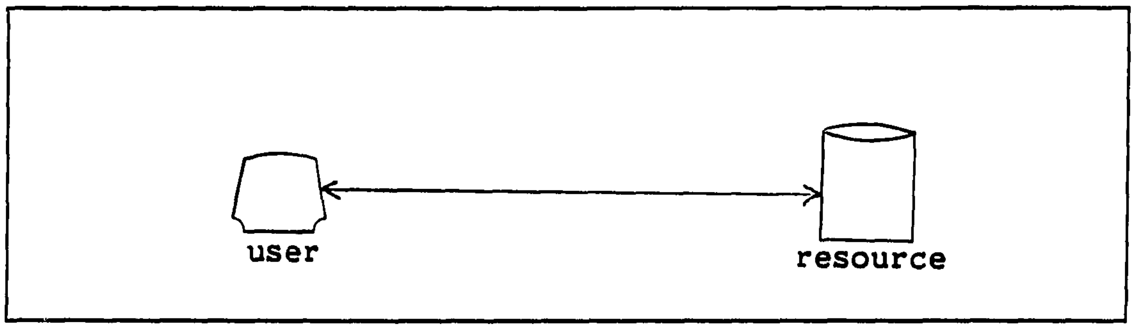

FIGURE 1. User's view in accessing a resource

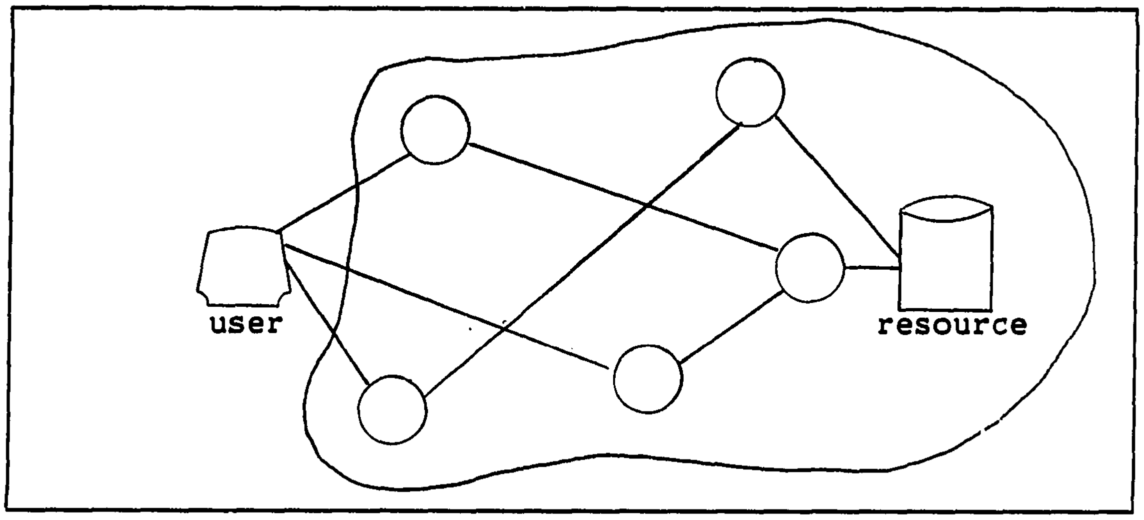

FIGURE 2. Transparent computer network

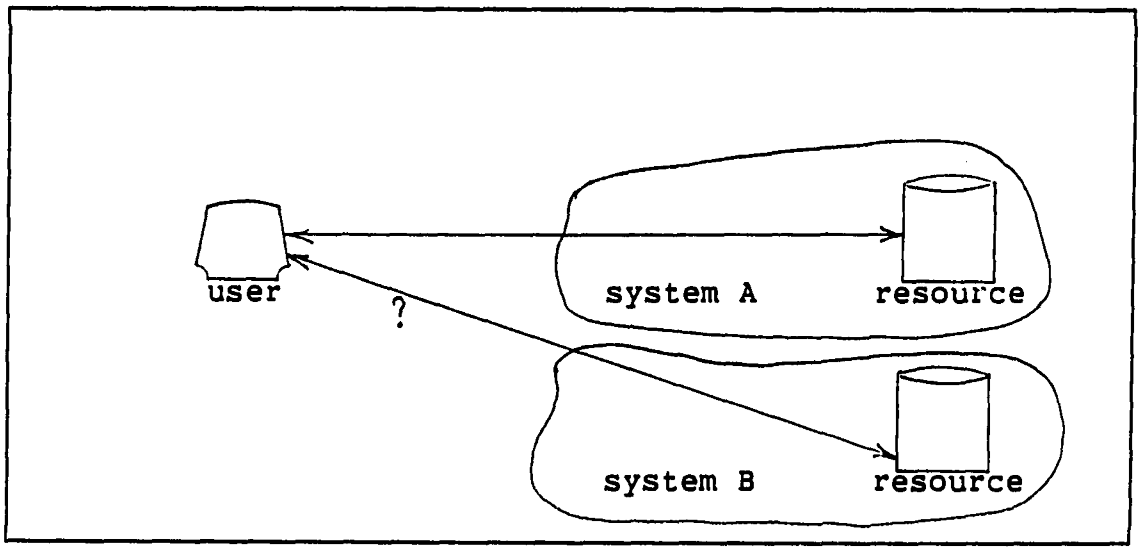

FIGURE 3. Problem of system adaptivity

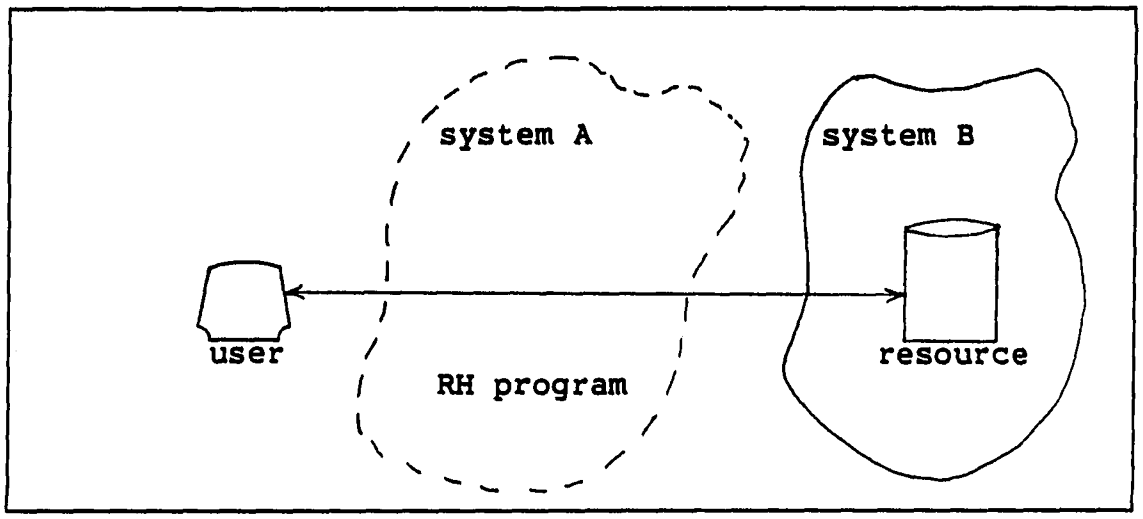

FIGURE 4. Remote host terminal emulator

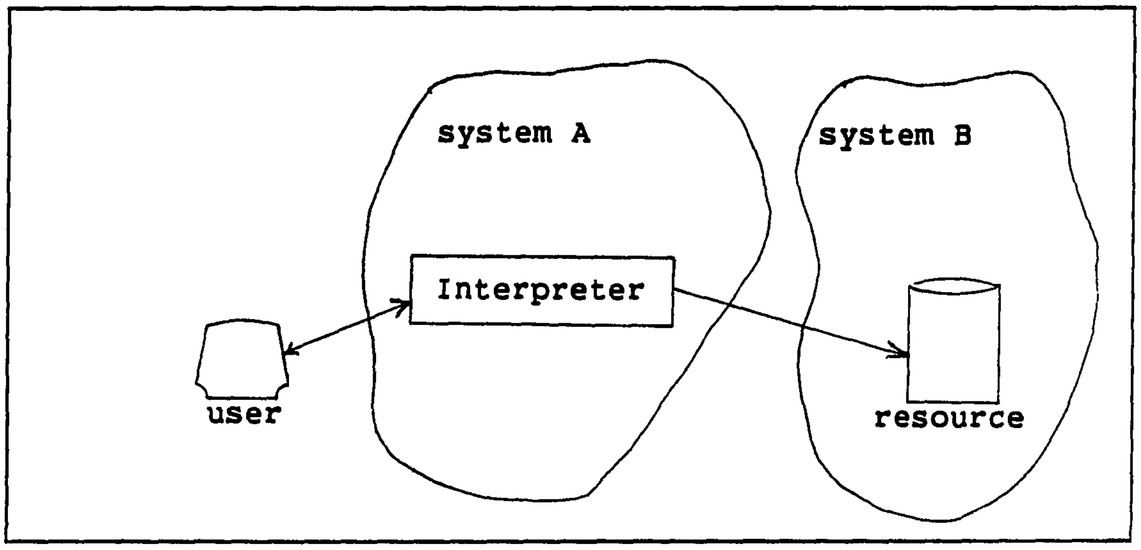

FIGURE 5. Command translation software

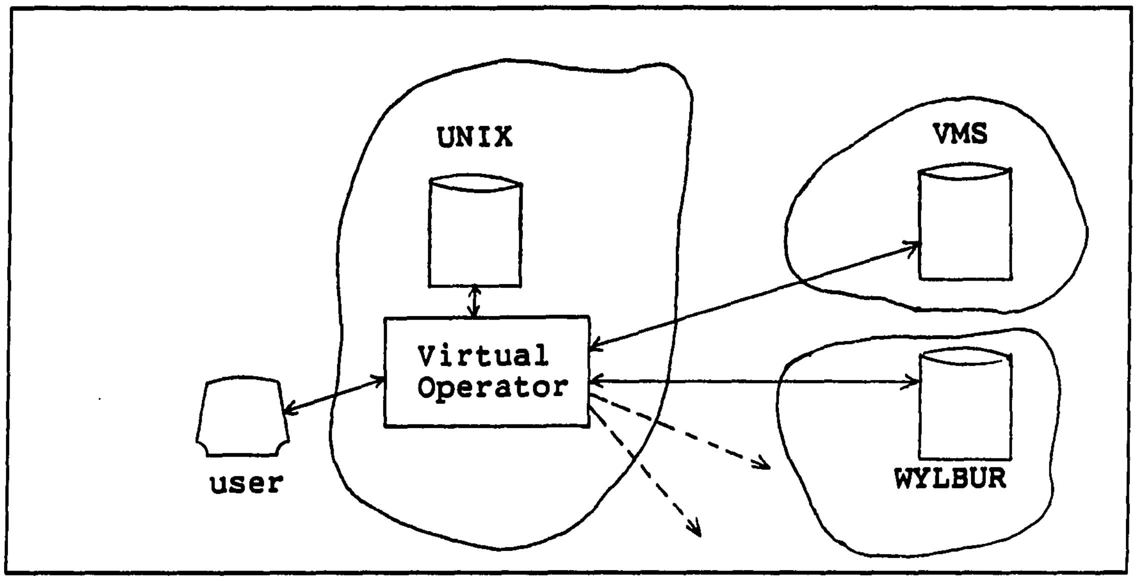

FIGURE 6. Virtual Operator approach

REVIEW OF LITERATURE

Over the past two decades, computer systems have been developed to simulate human activities, much of this activity being associated with the term Artificial Intelligence (AI) (Nilsson, 1980).

Since the middle of the 1970s, AI researchers have often turned to the limited field of specified problems instead of using a more general approach, usually because AI programming is much more difficult than more conventional programming in many aspects. The process of developing an AI program is relatively complicated because the program logic must not only describe a mathematical routine or routines as in conventional programs, but must also attempt to represent human intelligence factors such as reasoning and learning. Any such human quality, no matter how natural it may seem to us, is of course very difficult to formulate into logic understandable by a machine.

AI research, though complex, has been and is widely continued due to its acknowledged promise. In this chapter, the general progress of AI research and some fundamental concepts and theories are reviewed.

Applications of AI

Artificial Intelligence techiques have been applied in many different areas. Basically, those applications fall into the categories of Natural Language Processing, Computer Vision, Robotics, and Expert Systems (Gevarter, 1985).

Natural Language Processing

The goal of AI research in Natural Language Processing is to develop methods to interact with computers in natural language, which includes both language understanding and generation.

Computer Vision

Computer Vision is concerned with enabling a computer to identify what it sees, or to locate what it is looking for.

Robotics

Robotics is the field concerned with the intelligent connection of perception to action. The sensing includes vision, force and tactile sensing, and the sensing of the robot's internal state. The action is usually provided by mechanical arms, wheels or legs.

Expert systems

The concept of the expert system arose in the 1970s when AI researchers turned to the solution of narrowly focused real-world problems (Forsyth, 1986). Expert systems are capable of generating advice or making decisions in narrow subject areas, solving problems that usually require human expertise.

An expert system contains an Inference Engine, (which implements search and reasoning methods to find solutions), and a Knowledge Base, which stores the system's expertise (Forsyth, 1984). The process of building an expert system (often called Knowledge Engineering) involves interactions between the knowledge engineer and a human expert (Figure 7).

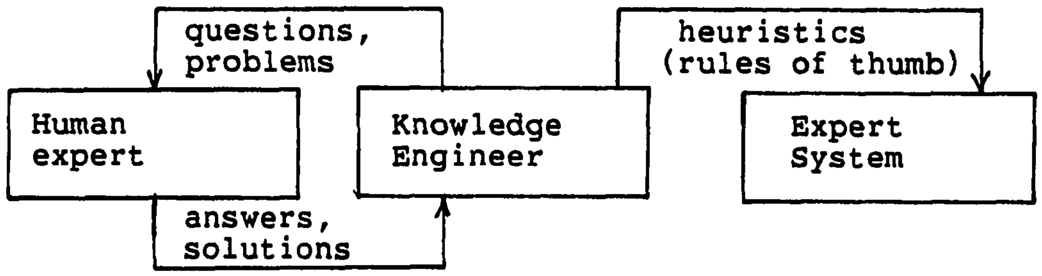

FIGURE 7. Process of Knowledge Engineering

Influences of AI on Programming

AI strategy strongly affects conventional programming. In this section, the influences and the progress of AI research are introduced by briefly characterizing several stages of programming methodology.

Stage 1: Conventional programming -- Do what I tell you to do.

Work is done by executing instructions one by one, each execution step being well defined in the program.

Stage 2: Heuristic reasoning -- Do according to the principles I gave you.

As AI techniques have been developed, AI programming has been used as a tool to support these new techniques, which are quite different from those based on Pascal-like programming. A program may be once again composed of instructions which explicitly determine the execution steps, but it is constructed by logically-defined induction rules and recursively applied to some known events to generate execution processes (Ramamoorthy, 1987).

This way to solve a problem is thus quite different; conventionally, we need a well-defined solution path, whereas now we only define the searching principles and let the machine generate the most reasonable solution path. This change in approach to problem solving makes it possible to produce a kind of machine reasoning by formulating human-like intelligence in logic.

Stage 3: Machine learning -- Improve yourself by experience.

Expert systems in their first stages can do jobs according to heuristics abstracted from human knowledge, but they will never improve because performance is not affected by experience, which is of course very unlike human behavior (Simon, 1983).

Theories of machine learning have been investigated for a long time, even before the invention of computers, but there was no efficient machine to fulfill the experiments at those times. With the improvement of computers, these theories have been reconsidered and used to develop expert systems, and new theories have also been developed through efforts of both mathematicians and psychologists.

It is the characteristic of expert systems in the second stage of development, that the machine can learn to improve itself, working better and better, and will avoid making a mistake twice. More details about learning theories are discussed in the next section.

Stage 4: Creative machine -- Create something I never thought.

To let a computer generate ideas that no one has ever thought of is the dream often fully exaggerated in scientific fiction. Expert systems in the second stage, though they may have learning ability, can not still deal with the situation when a solution is beyond the knowledge currently possessed. This is due to the fact that any derived rules are implied in the original rules, i.e., nothing "new" is created through the process of logic induction. It is obviously much more difficult to produce a machine which creates rather than learns; however, a few successful experiments featuring creative machines have been announced (Ritchie and Hanna, 1984). Research in this area is still very limited, basically because it is too difficult to find a model that can simulate such sophisticated phenomena in human behavior. Moreover, we do not understand fully enough the way in which the brain can think and invent to enable us to duplicate these functions.

Learning Theories

Most learning theories were initiated by psychological analysis of human behavior, formulated in mathematics, and then accomplished through computer simulations. Basically, they could be classified into three categories: learning by experiences, learning by being told, and learning by discovery.

Learning by experiences

A classified set of examples is provided to train the system, and the system develops its discrimination rules to do further classification jobs. The results after each decision are fed back as positive or negative instances to modify the discrimination rules. Many existing learning machines are based on this kind of methodology. One of the earliest systems, and still one of the best known, was the Perceptron (Rosenblatt, 1962). Training examples are represented by feature vectors, while discrimination rules are represented by a linear function composed of coefficients followed by feature variables. The learning process actually is a periodic sequence of parameter adjustment (Minsky and Papert, 1969), using mathematical optimization theory.

In some other learning theories, discrimination rules are represented in logic, which makes the rules developed more understandable. Training sets are also represented by a special description language, which makes the expression more comprehensive than that of using feature vectors.

Learning by being told

Learning by being told is the most straightforward method of learning. It provides a method to enable computer systems to acquire information from human experts representing knowledge in a specified domain.

The learning process proceeds in a question and answer manner (Grishman and Hirschman, 1978), and the system must keep track of interactive dialogue and know whether information is new or previously obtained in order to generate appropriate questions in the next question-answer iteration. Moreover, it must be able to remove incorrect information either according to instructions from users or through auto detection and correction mechanisms.

Learning by discovery

When scientists began attempts to make machines which not only reasoned and learned but also created, a new approach appeared. The knowledge formulated in a machine depends not only on training examples, but also on the mutation/combination of existing knowledge. The BEAGLE (Forsyth, 1981) was a pioneer in this approach in which the heuristic evolutionary rule breeder is derived from a genetic algorithm (Holland, 1975).

Eurisko (Lenat, 1982) is a pure discovery program, the learning process in which is absolutely by discovery and not by examples. It made a discovery, previously not conceived, for a new design of a NAND/OR gate, subsequently proven to be successful in manufacturing.

FUNDAMENTALS OF THE VIRTUAL OPERATOR APPROACH

In this chapter, a comparison with previous AI research is made to describe the fundamental differences and problems in the described research. Then the criteria for developing the virtual operator are presented.

A New Approach to AI Application

The virtual operator, motivated by the practical need for improved computer interactivity between different systems, is research that challenges the acknowledged limitation of using AI. Two characteristics of this research, in which AI techniques are applied in the area seemingly in violation of the principles of using AI and also with increased implementation difficulty, describe the major differences from other AI applications.

1. Command translation is almost a one-to-one mapping process.

In order for a system to communicate with another system, it is required to translate commands passed between systems. Since any given system's commands are all well formatted, the translation is usually almost unique, i.e., the translation is or close to a one-to-one mapping process.

In general AI research, AI techniques are basically applied to areas with a high uncertainty of solution, and the goal is to obtain a suggestion or indication which may lead to the most appropriate solution. In the case at hand, in contrast to general AI applications, instead of finding a better or best solution, finding a successful one is the goal. For example, the task of copying a file from A to B is translated to a VMS command as "COPY A B" and, even though another alternative such as "COPY A C, COPY C B" exists, we are not much interested in finding the better translation or the number of possible translations, as long as a valid one is found.

2. Accumulated experiences may not be useful.

Almost all AI research efforts are based on the assumption that intelligence is a result of knowledge accumulation. The AI chess player, for example, will memorize each new scheme or strategy from its opponent and therefore, through practicing, will improve and perhaps eventually become an expert.

Many computer systems are quite different from one another in their instruction set, and thus accumulated knowledge in translating commands for a given system may be almost useless in translating commands for another system. For example, knowing about a command such as "COPY A B" in VMS does not help to understand that the same command is "cp A B" in Unix. By analogy, an expert in chess may not be more proficient in learning poker than a person without any experience, since the playing schemes are too different. Knowledge accumulated thus may have no direct benefits if the new system is too different from the one in experience. Unfortunately, this is very often the case.

AI Strategies for Solving Problems of Computer Interactivity

Applying the above facts to the problem of computer interactivity, we come up with the conclusion:

> It is not appropriate to apply AI techniques to a command translation process in which no rules can be actually induced and used in another translation process.

This is a reluctant conclusion after a long search including both from theoretical and experimental approaches, but the work has not necessarily been in vain because the characteristics of intrinsic problems hidden in the realm of computer interactivity has been generally explored and better understood, making it possible to better apply AI strategy at a crucial point where this approach can effectively enhance computer-to-computer interactivity.

Strategic Principle 1:
Use the " Learning by Being Told " strategy in collecting commands of a new system.

For a new system, the best way to improve the knowledge base of the virtual operator is let a human expert tell whatever he knows about using the system, the virtual operator recording all the command sets.

Strategic Principle 2:
Use AI heuristic rules in the task analysis and command classification process, and help the virtual operator to accept a new system by inheriting and modifying previous experiences.

Experiences in translating commands cannot be exhaustively formulated due to the great variety of commands in different systems, but the experiences in classifying commands and relating them to a certain task can be formulated as heuristic rules to help the virtual operator to compose possible command sequences to provide the same service in a new system.

LEARNING SIMULATION

It is found by observing human behavior in dealing with new knowledge that background knowledge in experience is first assumed valid for the new environment, a process known as knowledge generalization. Exceptions may occur when the generalized knowledge is applied and found inappropriate, at which time specifications will be added to modify background knowledge in order to adapt to the new environment, a process known as knowledge specification (Lenat, 1982). Generally, knowledge integration is a combined effect of knowledge generalization and specification. Results of applying fundamental rules to a new domain will be fed back and used to refine previously existing knowledge; positive results cause the further generalization of the applied rules, whereas negative results make the rule more tightly specified. Figure 8 shows the flow chart for learning simulation.

Knowledge Classification

Two kinds of knowledge are required for the virtual operator to do the job as a self-improvement human operator. First is the knowledge of commands for providing services, including command syntax, argument description, or command sequence if the service is not a one-step service. This

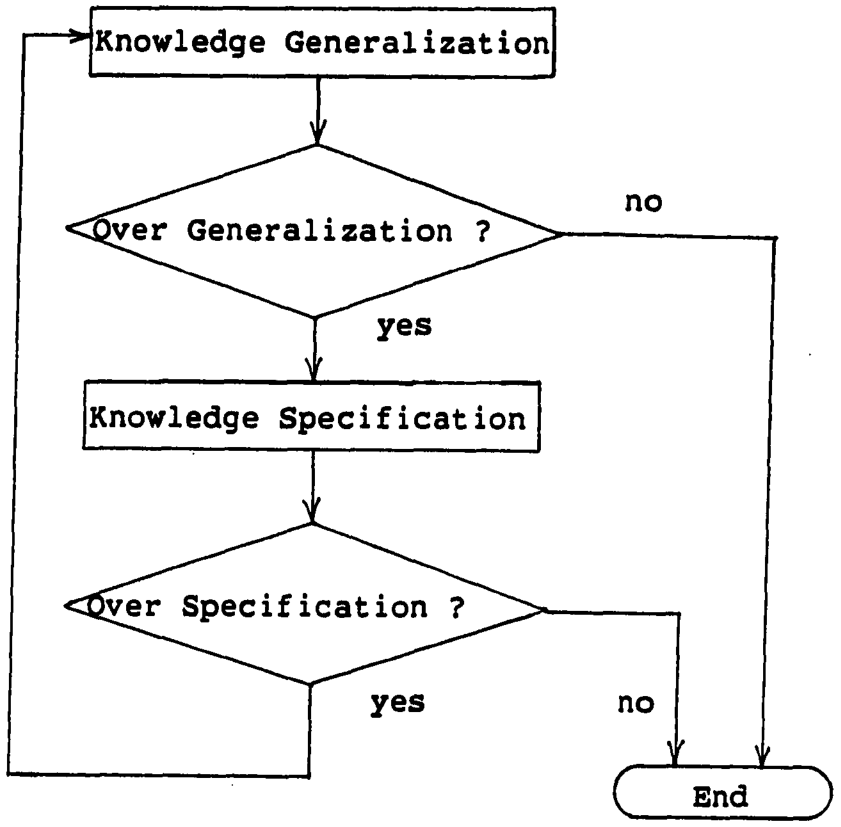

FIGURE 8. A flow chart for learning simulation

kind of knowledge is obtained through the process of learning by being told and saved in the form of command tables to provide a base of different system languages for communicating with different systems.

The second type of knowledge is a foundation of reasoning ability for giving suggestions for action when a requested service has not yet been implemented. This kind of knowledge, including complexity, possible command sequences, and the implementation status of a service, is represented by frame-like tables, which are continually updated and modified under the control of embedded heuristic rules. In terms of generality, this type of knowledge is further classified into two levels: global knowledge, general information among systems to characterize their "commonness", and local knowledge, special characteristics within a system to characterize its distinct differences. The virtual operator will continually evaluate each of the rules to adjust its generality, and will transfer knowledge between global and local knowledge bases by knowledge generalization and specification.

To communicate with a new system, the virtual operator will first use global knowledge and knowledge generalization, assuming that the knowledge is also valid for this system. The inherited knowledge will then be tried in the new system, and be rejected, accepted or modified through the process of knowledge specification.

Local and Global Service List

Generally-used utilities are commonly supported by different systems, but almost every system also has its own special utilities, representing features different from those of other systems. A single service list can only represent the commonness and not the distinctions of system characteristics, and therefore it is not a good knowledge representation for system characteristics. The suggested representation is a combination of two service lists consisting of a common service list, copied from the global knowledge base, appended to a special service list, saved locally in the local knowledge base. The common service list maintained in the global knowledge base (called hereafter the global service list) keeps collecting general services existing in some other systems, and supporting the information base of possible services to the system under training. The special service list contains special utilities unique to each other system, presenting the special features of each such system. The service list for a particular system (called hereafter the local service list) is based on this kind of combination.

Users can keep adding new services to the local service list, and unless the new service is specified "unique", it will be considered as a possible service for some other systems. It will then be added to the global service list in the global knowledge base, to be broadcast and appended to every other system's local service list, which the virtual operator uses to generate questions to get corresponding commands.

Over-Generalization

Over-generalization is the situation in which a generalized assumption is not true for a new environment. Considering the knowledge generalization process of the virtual operator, some service names in the local service list are inherited from the global knowledge base, which is based on the assumption that those services already present in some other systems are also available in the new system under training. Whenever this assumption is found invalid, over-generalization has occurred.

Messages transferred from the global knowledge base include service names and related statistical analyses for implementing those services. Whenever a training phase is initiated, the virtual operator will ask for command information to implement the service, while in the meantime suggestions based on statistical results will be generated to help the user. If the number of unsuccessful tries exceeds a pre-defined threshold, the virtual operator will treat this case as an over-generalization error and remove the service name from the local service list.

Over-Specification

Over-specification is the situation in which a rule could be used more generally than it is actually used. Over-specification will not cause an error as the case of over generalization, however, it reduces the power of using rules in organizing and simplifying knowledge construction.

A user can specify the knowledge of a service as "unique" to prohibit knowledge generalization and block the message transfer to the global knowledge base, which is a way to render a system distinct from others, but it may happen that a service specified as "unique" is in fact potentially applicable to another system. The virtual operator will also check these special services, and whenever they are also found to be applicable to another system, the operator will remove the restriction and transfer the information to the global knowledge base and then broadcast it to other systems.

The local service list is thus maintained for each system, and is expanded by adding new services either from human experts or by automatic knowledge inheritance from the global service list in the global knowledge base. In the other aspect, reducing by knowledge specification, the service name is removed whenever an over-generalization error is detected.

STRUCTURE OF THE VIRTUAL OPERATOR

Now that the general methodology of implementing a virtual operator has been described, we will take a closer look at the interior organization of the operator.

The basic elements of the virtual operator are the user interface, the heuristic center, and the transportation manager. They are described in the following sections and illustrated in the block diagram as in Figure 9.

The virtual operator has two operation modes, the training mode and the using mode. In the training mode, the virtual operator acts as a student and treats the user as a teacher. The user in this case should be a human expert able to teach the virtual operator to communicate with a particular system. In this mode, knowledge from the user is collected in the heuristic center and the transportation manager is disabled since no communication service is made.

In the using mode, the virtual operator acts as both a real operator and as an interpreter; commands and communication principles kept in the heuristic center are used by the transportation manager to perform a specified task.

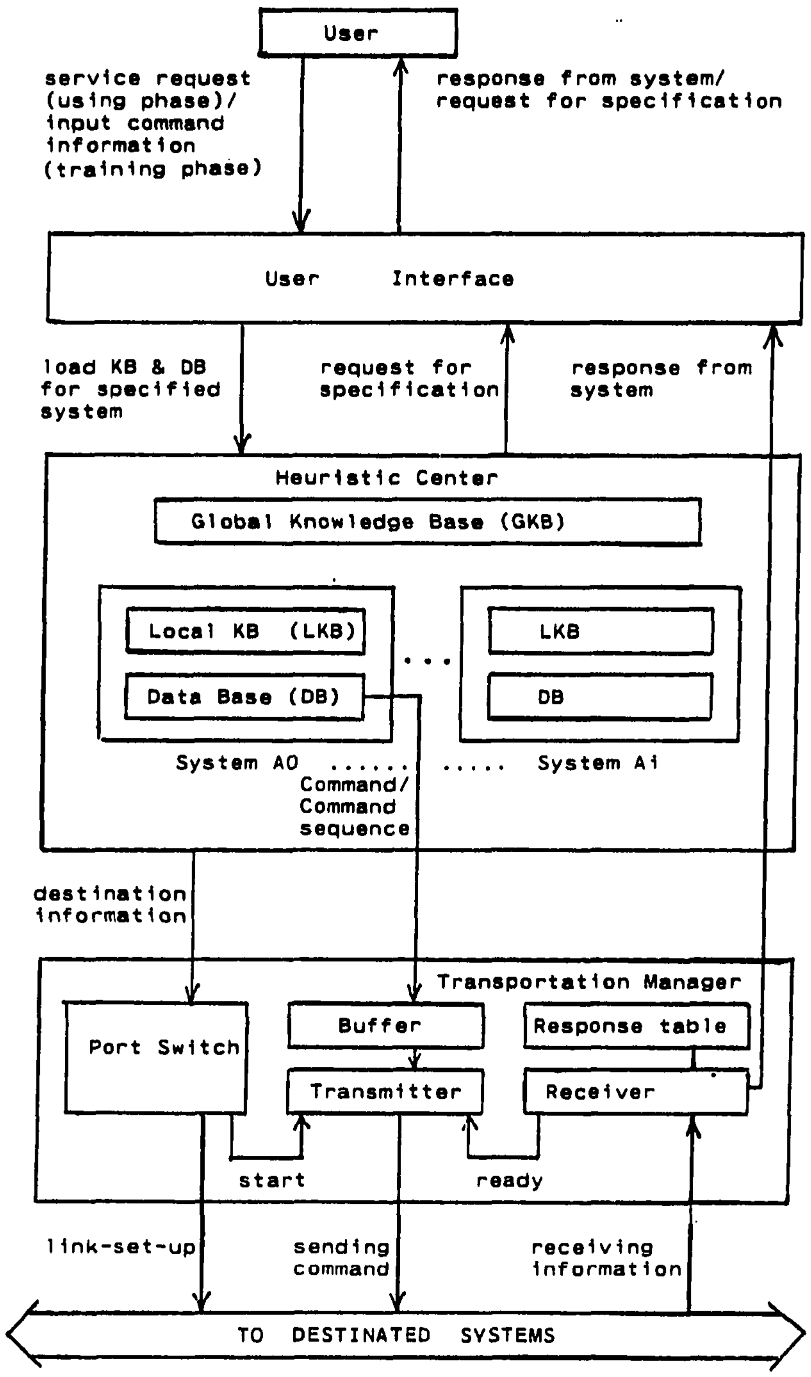

FIGURE 9. Structure of the virtual operator

User Interface

The user interface is a bridge between the user and the interior of the virtual operator. It makes the work of the operator transparent to the user, and the user therefore detects no variation when communicating with different systems.

The user interface is implemented as a user application program, written in Unix Shell commands, in which a service table is displayed to help the user in specifying an object system and selecting operation modes.

Heuristic Center

The Heuristic Center is the "brain" of the virtual operator, containing both a database and a knowledge base. The database is the place where command knowledge is stored, and it is basically composed of structured command tables. The knowledge base is somewhat more complicated. Although it is still a hierarchy of files as in the database, the information saved is compiled knowledge, instead of raw data or information.

Database

Commands and related information that support the virtual operator with versatile communication languages are saved in the database. A directory is assigned for each system, and all the information related to that system is kept under the specified directory. This information includes service numbers, a local service list, a command table, a number of command files, and a file name list.

<u>Service number</u> Whenever a new service is added to the global knowledge base, a service identification number is assigned to it. These numbers are kept in a sequential file corresponding to the local service list.

<u>Service description</u> Descriptions of the services in the system, both implemented and incomplete, are kept in a sequential file as a local service list.

<u>Command table</u> A table of commands is saved in a file in which all commands currently known to the virtual operator are listed. Any added command can be checked by scanning the command table to determine whether it is a new command or not.

<u>Command file</u> Command files are structured as frame-like tables, like the one shown in Figure 10, each with six formatted fields, in which the file name, the description of the service, the description of the arguments, the number of steps, and the command syntax are stored. The number of steps indicates the complexity of the service; if it is a single step service, the number of steps is one; if it is

not, the number of commands required to complete the service is recorded, and a command sequence instead of a command will be saved in the field of the command syntax.

Description of a service	Receive a file from Unix.
Number of arguments	2
Description of argument 1	SOURCE
Description of argument 2	DESTINATION
Number of steps	3
Command syntax 1	source %1
Command syntax 2	collect
Command syntax 3	save %2

FIGURE 10. The command file of a service in WYLBUR

<u>File name</u> The file name is the path name used to access a particular command file. These are also kept in a sequential file as a table of entries to unit of each utility information.

The file name of a single step service is made up of the first two characters of the command, appended to the corresponding service number. The file name of a multi-step service is just the sequence of service numbers that correspond to the sequence of commands required during the service.

Knowledge base

Except for the global knowledge base, saved in the home directory, all the local knowledge bases are saved locally under the specified directories along with their command files. The knowledge base still has a frame-like structure, as shown in Figure 11, and the frame for each service is divided into six fields respectively containing the service number, the description of the service, the number of systems in which the service has been implemented in one step, the number of systems in which the service has been implemented in more than one step, the number of heuristic rules, and contents of heuristics.

```
Service number              : 12
Description of service      : Receive a file from Unix.
Single command (number of systems) : 0
Multi commands (number of systems) : 2
Number of heuristic rules : 1
Heuristic ID                : 0
Weight                      : 2
Command sequence            : 9-10-3
```

FIGURE 11. Information of a service in the global knowledge base

The local knowledge base is essentially a partial projection of the global knowledge base, containing only information relative to incomplete services. Another region in the local knowledge base contains information about those special services which are unknown to the global knowledge base, representing unique features of a particular system. Several elements of miscellaneous information are maintained locally in the local knowledge base, e.g., a counter in which the number of failures to implement a service is recorded, and three implementation status variables, indicating whether the service is new, completed, or changed. Details of those functions are described in the following sections.

Command sequence suggestion Any successful command sequence used in performing a certain service will be recorded and used as a suggestion to implement the same service in another system. Accumulated experiences may produce more than one suggestion for a particular service, and the operator will keep updating the weight of each suggestion based on the evaluation function.

Suggestions will prompt the user in the order of calculated weights. The one with the biggest weight is considered to be the best suggestion by the virtual operator, and is always sent to the user first. The weight of a suggestion is determined by evaluation functions, which are defined and can be varied by the knowledge engineer (the one implementing the virtual operator) according to the heuristic information available. The number of successes, for example, can be used to define such an evaluation function. By setting the weight of a suggested command sequence equal to the number of successes in using the sequence to perform a service, the suggestion with the most successful experiences is established as the best one.

The number of failures can also be used as heuristic information to determine the evaluation function. By setting the weight equal to the negative of the number of failures, the suggestion with the least failures is established as the best one.

The "most success" and the "least failure" methods are the simplest evaluations, but they both suffer from the vague definition of success; neither of them necessarily presents the true probability of success. A more reasonable evaluation would calculate the probability of success by setting the weight equal to the percentage of success with respect to the number of total tries.

When a new system under training is in the same manufacturer's series as a previously experienced system, and when the two have a strong resemblance to one another, evaluation functions based on statistical results may lead in a wrong direction. For this special case, we should ignore the statistical results and take the suggestion directly from the experiences of the similar system. Taking the resemblance factor into consideration, the suggested evaluation function for a suggestion drawn from communication experiences with system A is:

> a*(number of success)/(number of total tries) + b*1/((ID of system A - ID of the system under training)+1)

where a,b are adjustable coefficients to control the weight of each factor. Numeric difference between system IDs will be chosen to represent the resemblance between systems. A summary of the evaluation functions is given in Table 1.

TABLE 1. Evaluation Function of the suggestion from system A

HEURISTIC INFORMATION	EVALUATION FUNCTION
Most success	f_0= Number of successes
Least failure	f_1= - (Number of failures)
Probability of success	f_2= No. of successes/Total Tries[a]
Resemblance	f_3= $1/(\lvert ID_a - ID_{training} \rvert + 1)$
Combined[b]	f_4= a*f2 + b*f3

[a]Total tries is the sum of the number of successes and the number of failures.

[b]Adjustable variables a and b.

<u>Service complexity</u> For each service, the number of systems that have completed the service in a single step and

the number of systems that have completed the service in more than one step are recorded in the knowledge base. This information will help the user to decide whether or not to accept a suggestion of trying a command sequence in order to accomplish the service. For instance, if the statistical record shows that the service is 90% likely to be a single command service, the user had better try to find the particular command instead of trying any other suggested command sequences.

Failure count To prevent the over-generalization error mentioned previously, a counter for each incomplete service is used to accumulate the number of failures in obtaining command information. A default threshold can be set, and whenever the number of failures exceeds this value, the virtual operator will consider it as an over-generalization case and stop further inquiry by removing the service name from the local service list.

Effective heuristics To prevent the virtual operator from making any given mistake twice, any heuristic tried and failed shall not be used again. The virtual operator will keep information about use of a heuristic in the local knowledge base to determine whether the heuristic is still considered effective or not.

Status variables and updating rules   Three implementation status variables for each system are saved along with its local knowledge base. A status variable is composed of an array of sixteen bit integers and manipulated under the bit operation supported in C language, each bit indicating the current status of a corresponding service (the nth bit corresponding to the service with a service number n). For example, the least significant bit indicates the status of the service with a service number zero.

The first status variable "New", indicates a service in the global knowledge base which is new to the system under training. A bit in this variable is set when there is a new service added into the global knowledge base and reset after the corresponding information of this new service is copied to the local knowledge base. Figure 12 and Figure 13 show the condition before and after knowledge transfer.

```
Status variable "New":  0000000000000001

Local knowledge base : Empty
```

FIGURE 12. Local knowledge base before knowledge transfer

The second status variable, "Complete", indicates which service in the global knowledge base has been implemented in

```
Status variable "New":  0000000000000000

Local Knowledge base :

Service number              : 0
Description of a service    : Display the file names.
Single command : 2
Multi commands : 0
Number of heuristic rules : 0
```

FIGURE 13. Local knowledge base after knowledge transfer

the system under training. A bit in this variable, initially reset, will be set when the corresponding service is implemented and available for the using phase.

The third status variable, "Change", indicates which service information in the global knowledge base has been changed. A bit in this variable will be set when a change in the statistical results of the corresponding service is detected in the global knowledge base, and is reset when the updated information is copied to the local knowledge base.

At the beginning of each training phase, for those services with a corresponding bit set in the variable "New", the virtual operator appends all information, including the service number and descriptions, from the global knowledge base to its local knowledge base. For those services with a corresponding bit set in the variable "Change" but a bit reset in the variable "Complete", the virtual operator replaces the statistical information in the local knowledge base with the latest information from the global knowledge base. The updating rules described are shown in Figure 14.

Transportation Manager

The actual communication work, such as link set-up and data-flow control, is done by the transportation manager of the virtual operator. The transportation manager knows nothing about commands but is familiar with different transportation protocols of different systems. Basic parts contained in the transportation manager are the transmitter, the receiver and the port switch.

Log on procedure

In the using phase, whenever a service is requested, the transportation manager starts communication by initiating the log on procedure to the requested system. The port switch first connects the user's terminal to the system by assigning the user a logical channel which is a link port to the requested system. Then the transmitter sends log on information to the connected system and waits to be identified. In the meantime, the receiver is also activated by the transportation manager; it monitors the response from the connected system. There is a small database built into the receiver to keep some key words,

```
Ri: System that is currently under training.
R: Systems that are not under training.

Initialization:
IF a bit in the status variable "New" is set,
   (i.e., a service that is new in Ri.)
THEN
1. Add the service name to the local service list.
2. Add the characteristic information of the service
   to the local knowledge base.
3. Reset the bit in "New" of Ri.
4. Reset the bit in "Change" of Ri.
5. Reset the bit in "Complete" of Ri.

ELSE IF a bit in the status variable "Change" is set
        but is not set in the "Complete",
        (i.e., the information of an incomplete service
               has been changed.)
THEN
1. Replace the characteristic information of the service
   in the local knowledge base with the updated
   information in the global knowledge base.
2. Reset the bit in the status variable "Change" of Ri.


End of the training phase:
IF a new service has been added AND
   it is not specified as a unique service.
THEN
1. Add the information of the service to the global
   knowledge base.
2. Set the bit in the status variable "Complete" of Ri.
3. Set the bit in the status variable "New" of R.

ELSE IF an original incomplete service is completed.
THEN
1. Update the information in the global knowledge base.
2. Remove the information of this service from the local
   knowledge base.
3. Set the bit in the status variable "Complete" of Ri.
4. Set the bit in the status variable "Change" of R.
```

FIGURE 14. Updating rules

such as the system prompt and error messages generated by the system for indicating status. The receiver can therefore understand the meaning of received messages and take further action.

The log on procedure generally is terminated by receipt of the system prompt, which indicates that the user has been logged onto the system and the system is ready for accepting commands from the user. A signal will be sent to the transmitter by the receiver to indicate this situation.

Data-flow control

There is a command buffer built into the transmitter to hold the command or the command sequence to be sent. Actually, before the transportation manager initiates a log on procedure, the heuristic center would have prepared the command or the command sequence (according to the requested service) and transferred it to the command buffer. When the log on procedure is completed, the transmitter receives a signal from the receiver and starts to send the command.

In most cases, the receiver is scanning the response and waiting for the system prompt, which is the indication for sending the next command or data element. Whenever this indication is detected by the receiver, a ready-for-transmit signal is sent to the transmitter. The communication is processed under this kind of data flow control, and data transmission is thus always continued at the most efficient rate without allowing any data to be missed.

End of session

After the requested service is completed, the transportation manager will initiate the log out procedure to disconnect the logical link. The service list is displayed again and the user can choose another service or exit the using phase.

Implementation of the transportation manager

The transportation manager is implemented by fully utilizing the characteristics of the Unix operating system and its versatile system calls. Any system physically connected to Unix is considered as a special device and can be accessed just like any ordinary file. The transmitter and receiver are two concurrent processes written in C language and using Unix system calls. Interprocess communication is implemented through a "pipe", via which the receiver sends the ready-to-transmit signal to the transmitter.

<u>Port switch</u> Any system connected to Unix is assigned a special device name as the logical path to its access. In the using phase, the virtual operator will first ask the user to specify the system to which he wants to

connect, then display the available services to the user. With a specified system name, the virtual operator finds the corresponding logical name and commands the port switch to set up the link.

The channel lock mechanism is based on a semaphore technique. Before a particular channel is assigned to a user, a semaphore is checked to see whether the channel has been occupied or not. If it has been occupied, another channel will be checked. Otherwise, the destined system is connected through this channel, and the semaphore is set to prevent any other user from interrupting during the conversation.

Transmitter The transmitter is implemented as a process in which data or commands prepared in the command buffer are written to the destined system. The transmitter process does not analyze the response from the connected system, but it generates a concurrent child process, the receiver, for this purpose. A state diagram of the transmitter is in Figure 15.

Receiver The child process of the transmitter, the receiver, keeps collecting messages from the connected system. By interpreting key words saved in its database, it is capable of determining the status of the system and informing the transmitter as to what is going on. A state diagram of the receiver is in Figure 16.

<u>Ready-to-transmit</u> <u>signal</u> The interprocess communication relies on a one way pipe (produced by a Unix system call) from the receiver to the transmitter. A special character is used as the ready-to-transmit signal and sent through the pipe, and the transmitter process will not send the next command or data element until the ready-to-transmit signal is received from the pipe.

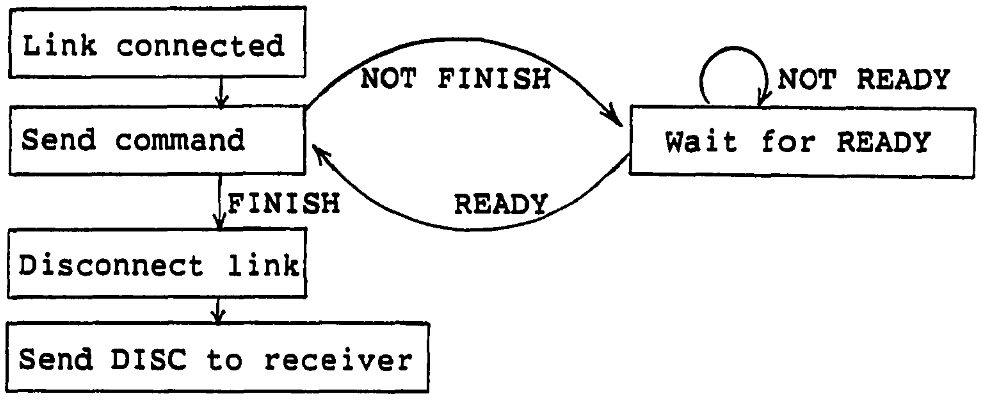

FIGURE 15. A state diagram of the transmitter

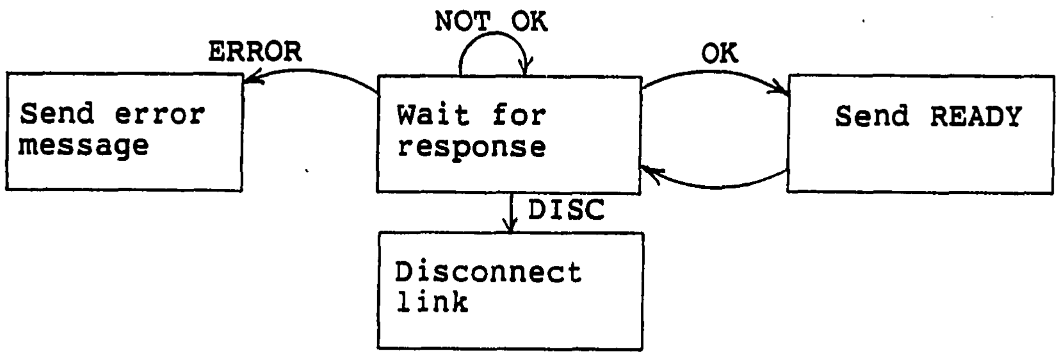

FIGURE 16. A state diagram of the receiver

USER VIEW

The virtual operator has two operation phases from the user's point of view. One is the training phase, in which the operator learns and accepts new communication knowledge from users; the other is the using phase, in which the operator serves users as a human operator, setting up the logical link, and translating and sending the user's commands to the requested system.

The Training Phase

The training phase is initiated by special users, such as human experts, for improving the knowledge base of the virtual operator.

During the training phase, the virtual operator acts as a student, and treats the user as a teacher. It is built to be both a good questioner and analyst, and thus all the user needs to do is to answer questions by typing the command syntax and related information, not worrying about how the operator is collecting knowledge. The operator will use its own internal knowledge representation to organize the received information and abstract the raw information into heuristic rules.

The following is a sample of dialog to illustrate the process of knowledge integration in two iterations of the training phase. In the first iteration, shown in Figure 17, VMS was chosen as the object system with which to teach the virtual operator to communicate. Since the knowledge base is empty at this point, the virtual operator can do nothing but ask the user to input service information such as the description of a service and its arguments, the number of arguments, the command syntax, the command sequence, and the generality of the service. In the second iteration, shown in Figure 18, WYLBUR was chosen as the object system to learn. Since the virtual operator has now been trained to communicate with VMS, all the accumulated experiences may be used to generate heuristic suggestions to help the user. It can be seen in the beginning of the second training iteration, by comparison with the first iteration, that the virtual operator knows that there could be a service which displays file names, and the probability that this service could be implemented in a single WYLBUR command is one. These suggestions are based on the previous experience in communicating with VMS; the idea of displaying file names, originally unknown to the virtual operator, is implemented by a single command in VMS, therefore it suggests the user that he may also implement this service in WYLBUR, and the probability to find a single command to achieve this is one, because so far the virtual operator has never met the case that a system needs more than one step to accomplish this service.

How to receive a file from Unix, as in the second example, is learned by the virtual operator in the first training phase with VMS. The virtual operator displays the most reliable command sequence (actually the only sequence it has learned so far) to implement this service in the training phase of WYLBUR, and waits for the user to make a decision as to whether to take the suggestion or not. During the training phase, the virtual operator will always be a student; it will generate questions or suggestions but will never make the final decision for the user, except when he gives up the opportunity.

One of the questions generated by the virtual operator in the training phase is about the generality of a service. Whenever a new service is defined by the user, he is responsible for telling the virtual operator whether he thinks this service is also available in another system or not. If his answer is yes, the related information learned in implementing this service will be applied to some other training phases with other systems; otherwise, those experiences will be marked as special techniques and only kept in the local knowledge base for the particular system as one of its special features.

```
1: Wylbur
2: Vms
3: Unix
4: EXIT
INPUT SYSTEM ID>
2

1: Training Mode
2: Using Mode
3: EXIT
INPUT MODE NUMBER>
1

******************************************
ADD NEW COMMANDS (y/n)? y

ONE STEP COMMAND (y/n)? y
INPUT THE SYNTAX, USING %1, %2.. AS ARGUMENTS >dir

INPUT THE DESCRIPTION OF THIS COMMAND
>Display file names.

IS THIS COMMAND ONLY SUPPORTED IN THIS SYSTEM? (y/n)>n
INPUT THE NUMBER OF ARGUMENTS >0


******************************************
ADD NEW COMMANDS (y/n)? y

ONE STEP COMMAND (y/n)? y
INPUT THE SYNTAX, USING %1, %2.. AS ARGUMENTS >e

INPUT THE DESCRIPTION OF THIS COMMAND
>Save a file.

IS THIS COMMAND ONLY SUPPORTED IN THIS SYSTEM? (y/n)>n
INPUT THE NUMBER OF ARGUMENTS >0


******************************************
ADD NEW COMMANDS (y/n)? y

ONE STEP COMMAND (y/n)? y
INPUT THE SYNTAX, USING %1, %2.. AS ARGUMENTS >source %1

INPUT THE DESCRIPTION OF THIS COMMAND
```

FIGURE 17. The training phase of VMS

```
>Built in command for specifying source file during file
 transportation.

IS THIS COMMAND ONLY SUPPORTED IN THIS SYSTEM? (y/n)>n
INPUT THE NUMBER OF ARGUMENTS >1
INPUT THE DESCRIPTION OF ARGUMENT 1 >SOURCE


*******************************************
ADD NEW COMMANDS (y/n)? y

ONE STEP COMMAND (y/n)? y
INPUT THE SYNTAX, USING %1, %2.. AS ARGUMENTS >sos %1

INPUT THE DESCRIPTION OF THIS COMMAND
>Create a file.

IS THIS COMMAND ONLY SUPPORTED IN THIS SYSTEM? (y/n)>n
INPUT THE NUMBER OF ARGUMENTS >1
INPUT THE DESCRIPTION OF ARGUMENT 1 >FILE NAME


*******************************************
ADD NEW COMMANDS (y/n)? y

ONE STEP COMMAND (y/n)? n

INPUT THE DESCRIPTION OF THIS FUNCTION
>Receive a file from the Unix system.
INPUT THE SYNTAX, USING %1, %2.. AS ARGUMENTS >source %1

FINISHED (y/n)?>n
INPUT THE SYNTAX, USING %1, %2.. AS ARGUMENTS >sos %2

FINISHED (y/n)?>n
INPUT THE SYNTAX, USING %1, %2.. AS ARGUMENTS >e

FINISHED (y/n)?>y

TASK DESCRIPTION:  Receive a file from the Unix system.
COMMAND SEQUENCE
source %1
sos %2
e
INPUT THE NUMBER OF ARGUMENTS>2
INPUT THE DESCRIPTION OF ARGUMENT 1>SOURCE
INPUT THE DESCRIPTION OF ARGUMENT 2>DESTINATION
```

FIGURE 17. (continued)

```
1: Wylbur
2: Vms
3: Unix
4: EXIT
INPUT SYSTEM ID>
1

1: Training Mode
2: Using Mode
3: EXIT
INPUT MODE NUMBER>
1


******************************************
Display file names.

THE PROBABILITY IS 1.000000 FOR A SINGLE STEP SERVICE
ONE STEP COMMAND (y/n)? y
INPUT THE SYNTAX, USING %1, %2.. AS ARGUMENTS >show cat
INPUT THE NUMBER OF ARGUMENTS >0


******************************************
Save a file.

THE PROBABILITY IS 1.000000 FOR A SINGLE STEP SERVICE
ONE STEP COMMAND (y/n)? y
INPUT THE SYNTAX, USING %1, %2.. AS ARGUMENTS >save %1
INPUT THE NUMBER OF ARGUMENTS >1
INPUT THE DESCRIPTION OF ARGUMENT 1 >FILE NAME


******************************************
Built in command for specifying source file during file
transportation.

THE PROBABILITY IS 1.000000 FOR A SINGLE STEP SERVICE
ONE STEP COMMAND (y/n)? y
INPUT THE SYNTAX, USING %1, %2.. AS ARGUMENTS >source %1
INPUT THE NUMBER OF ARGUMENTS >1
INPUT THE DESCRIPTION OF ARGUMENT 1 >SOURCE


******************************************
```

FIGURE 18. The training phase of WYLBUR

```
Create a file.

THE PROBABILITY is 1.000000 FOR A SINGLE STEP SERVICE
ONE STEP COMMAND (y/n)? n
INPUT THE SYNTAX, USING %1, %2.. AS ARGUMENTS >collect
INPUT THE NUMBER OF ARGUMENTS >0


******************************************
Receive a file from the Unix system.

THE PROBABILITY IS 0.000000 FOR A SINGLE STEP SERVICE
ONE STEP COMMAND (y/n)? n

SUGGESTED COMMAND SEQUENCE 0:

FUNCTION : Built in command for specifying source file
           transportation.
SYNTAX   : source

FUNCTION : Create a file.
SYNTAX   : collect

FUNCTION : Save a file.
SYNTAX   : save
INPUT THE SYNTAX, USING %1, %2.. AS ARGUMENTS >source %1

FINISHED (y/n)?>n
INPUT THE SYNTAX, USING %1, %2.. AS ARGUMENTS >collect

FINISHED (y/n)?>n
INPUT THE SYNTAX, USING %1, %2.. AS ARGUMENTS >save %2

FINISHED (y/n)?>y

TASK DESCRIPTION:  Receive a file from the Unix system.
COMMAND SEQUENCE
source %1
collect
save %2
INPUT THE NUMBER OF ARGUMENTS>2
INPUT THE DESCRIPTION OF ARGUMENT 1>SOURCE
INPUT THE DESCRIPTION OF ARGUMENT 2>DESTINATION
```

FIGURE 18. (continued)

The Using Phase

The using phase is provided for general users, even those who have little computer background, for serving them like a human operator, taking care of all communication procedures. A service table will be displayed to provide the user with information about services currently available through the operator, and the user only need to select a service number to access a given utility.

The following sample dialog, shown in Figure 19, illustrates the using phase of WYLBUR, in which a file is transferred from Unix to Wylbur. Another example (Figure 20), shows the using phase of Unix. The user needs to specify file names to indicate the source and destination then the virtual operator will take over and transfer the file as the user has requested.

```
1: Wylbur
2: Vms
3: Unix
4: EXIT
INPUT SYSTEM ID>
1

1: Training Mode
2: Using Mode
3: EXIT
INPUT MODE NUMBER>
2
1 :Display file names.
2 :Retrieve a file to the active file.
3 :List the content of the active file on screen.
4 :Delete a file.
5 :Save a file.
6 :Print the active file to the output queue.
7 :Display the users currently logged on.
8 :Display the current date and time.
9 :Create a directory.
10 :Built in command for specifying source file
    during file transportation.
11 :Create a file.
12 :Display a file on screen.
13 :Print a file.
14 :Copy a file between same systems.
15 :Receive a file from the Unix system.

INPUT THE SERVICE NUMBER>15

*************************************************
COMMAND SYNTAX
source %1
collect
save %2

SEND COMMAND (y/n)?y
ARGUMENT 1 : SOURCE
 INPUT>cpf1
ARGUMENT 2 : DESTINATION
 INPUT>sample
```

FIGURE 19. The using phase of WYLBUR

```
ENTER SYSTEM ID:

Connecting to port 11A-BA003

Iowa State University Computation Center
User? nl.ycw
Password?
Account I6640?
Last logoff at 11:41 09/21/87
Last password change at 16:18 09/16/83 (1467 days ago)
Command> collect
    1.   > This is a sample file for testing the file
           transportation.
    2.   > asdfghjklzxcvbnmqwertyuiop
    3.   > 1234567890
    4.   > !@#$%¢_+¬|-=¢\{}<>?/:;"'
    5.   > ***
Command> save sample
SAMPLE saved on UCC005
Command> logout
GOOD BYE

**************************************************
1 :Display file names.
2 :Retrieve a file to the active file.
3 :List the content of the active file on screen.
4 :Delete a file.
5 :Save a file.
6 :Print the active file to the output queue.
7 :Display the users currently logged on.
8 :Display the current date and time.
9 :Create a directory.
10 :Built in command for specifying source file during
    file transportation.
11 :Create a file.
12 :Display a file on screen.
13 :Print a file.
14 :Copy a file between same systems.
15 :Receive a file from the Unix system.

INPUT THE SERVICE NUMBER>

1: Training Mode
2: Using Mode
3: EXIT
INPUT MODE NUMBER>
3
```

FIGURE 19. (continued)

```
1: Wylbur
2: Vms
3: Unix
4: EXIT
INPUT SYSTEM ID>
3

1: Training Mode
2: Using Mode
3: EXIT
INPUT MODE NUMBER>
2
1 :Display file names.
2 :Display a file on screen.
3 :Delete a file.
4 :Save a file.
5 :Rename a file.
6 :Print a file.
7 :Display the users currently logged on.
8 :Display the current date and time.
9 :Create a directory.
10 :Create a file.
11 :Copy a file between same systems.
12 :Send a file by mail.
13 :Communicate other users on line.

INPUT THE SERVICE NUMBER>1

**************************************************
COMMAND SYNTAX
ls

SEND COMMAND (y/n)?y

SENDING COMMAND TO UNIX.
:w3      current  functab  lp5      mv4      rm2
cal      da7      lkb      ls0      ph17     senddbs
compose  funcfl   lkb2     mal6     res      serv_no
cp11     funcls   lkbt     mk8      resdbs   syntax

**************************************************
```

FIGURE 20. The using phase of Unix

```
1 :Display file names.
2 :Display a file on screen.
3 :Delete a file.
4 :Save a file.
5 :Rename a file.
6 :Print a file.
7 :Display the users currently logged on.
8 :Display the current date and time.
9 :Create a directory.
10 :Create a file.
11 :Copy a file between same systems.
12 :Send a file by mail.
13 :Communicate other users on line.

INPUT THE SERVICE NUMBER>

1: Training Mode
2: Using Mode
3: EXIT
INPUT MODE NUMBER>
3
```

FIGURE 20. (continued)

CONCLUSIONS

Due to the improvement of computer architecture, it has become more and more feasible to apply artificial intelligence (AI) techniques in building expert systems. However, there are still too many problems in knowledge representation and integration to effectively bridge over the gap from human knowledge to machine intelligence.

How to transform human understanding in a certain area to a machine-acceptable representation is a very general problem. A principal difficulty resides in the difference of memory structure between human being and machines. Human intelligence is versatile, flexible, and free formed whereas a machine is based on a specified, fixed and structured construction. Moreover, the human brain is an incredibly powerful computer, each neuron in which is connected to thousands of others, making many human reactions virtually effortless. It is almost impossible for conventional sequential machines to simulate such sophisticated human behavior, and this capability is very limited even using modern techniques such as parallel processing in multiprocessor machines. The problem of knowledge representation becomes even more severe with increasing size of an application domain. Therefore, the AI approach has tended to be limited in application to more narrowly specified areas to reduce the complexity of knowledge representation.

The virtual operator, motivated by practical needs for interactivity between different computer systems, is a research task that has challenged these acknowledged limitations of using AI by applying AI techniques in a area that seems to violate the previous principles of using AI and to promise increased difficulty in implementation. In previously described research, AI techniques have been applied to a specified domain, and the goal has been to produce a machine-based expert of relatively narrow scope to perform a special task. The virtual operator research, while still oriented toward implementing a special-purpose expert system, enhances computer-to-computer interactivity by building a system that can learn to communicate with different systems. However the conceptual spirit is quite different, since the goal is to communicate with different systems, and to effectively share experiences among different domains.

To learn to communicate with a particular system represents narrowly specified knowledge, and to build a virtual operator that can learn to communicate with different systems actually expands this narrow learning to a much more general domain. Many problems were encountered in achieving this result, the most difficult one being how to efficiently use the experiences of communicating with one system in achieving an effective communication process with another system. Classifying knowledge in order to prevent worthless knowledge integration is a more severe problem than usually found in general AI research, because it is usually assumed that AI strategies are applied within a single specified domain, and accumulated knowledge is always considered useful and could be used in formulation of principles for that particular domain. This is not true if the knowledge domain is not unique. The virtual operator addresses the problems of using different domain knowledge from different systems. Knowledge of commands, for example, is found to be worthless in knowledge integration, since there are no rules which can be abstracted from commands of different systems.

A "learning by being told" technique has been adopted in the training phase of the virtual operator to acquire information about commands or command sequences used for communicating with different systems. General heuristic information and rules such as those command sequences are separated from the knowledge of command syntax by separating the knowledge base from the data base. Information about a command itself, such as the description of arguments and the command syntax, is saved in the database and retrieved as particular data in performing a certain task. Information about the characteristics of a service, such as a statistical survey about the varieties of command sequences or the number of steps to complete the service in different systems is saved in the knowledge base and used as heuristic information to generate suggestions to help the virtual operator in accepting a new system.

Learning processes are based on generalization and specification iteration, which is implemented by further classifying knowledge into local and global knowledge, representing both special and general rules for handling communication among different systems.

The virtual operator has been trained to communicate with VAX/VMS, VAX/UNIX, and NAS9160/WYLBUR. In the using mode, a user without significant computer background can use those systems through the help of the virtual operator, while in the training mode, a special user such as a person with expertise on a particular computer system can teach the virtual operator what he knows and thereby can install knowledge in the virtual operator.

In addition to serving the user much like a real operator, the virtual operator can also be used as a machine-based tutor to teach a user to manipulate a system with which he is not familiar. In this case, a real connection to a system may not be necessary, with only the command or the command sequence being displayed to the user according to the specified service description and the destined system name.

The heuristic power of generating suggestions in the training mode depends on how many systems the virtual operator has previously been dealing with; the more systems learned by the virtual operator the easier we train it to accept another new system.

Satisfactory operation in the using mode of the virtual operator also depends on the knowledge accumulated in its "brain". Therefore, how it is trained significantly affects overall performance. For a new service, the virtual operator relies on the user to give the service description, the argument description, and the command syntax. The special user training the virtual operator can check functions of new commands by trying them in a using mode, then detect and correct incorrect commands, but a vague or inappropriate description of a service function or its argument can not be "noticed" by the virtual operator, which may cause misunderstanding to the subsequent users.

The solution for this problem may depend on work in the area of Natural Language Processing, in which the methods of understanding and analyzing human language are concerned. A service classification routine might also be implemented to generate a service directory to help users more easily in locating their desired service.

In general, the research successfully introduced the idea of knowledge classification and representation for a problem related to different domains, leading to a step toward pushing AI strategies into a more general application.

BIBLIOGRAPHY

Charniak, Engene and McDermott, Drew. Introduction to Artificial Intelligence. Addison-Wesley, Reading, Mass., 1985.

Forsyth, R. "BEAGLE - A Darwinian Approach to Pattern Recognition". Kybernetes, 10, No. 3 (1981):159-166.

Forsyth, R. "The Architecture of Expert Systems". In Expert Systems, ed. R. Forsyth, pp. 9-17. Chapman and Hall, New York, 1984.

Forysth, R. "The Anatomy of Expert Systems". In Artificial Intelligence Principles and Applications, ed. M. Yazdani, pp. 186-197. Chapman and Hall, New York, 1986.

Forsyth, R. and Rada, R. Machine Learning. Halsted Press, New York, 1986.

Foxley, E. UNIX for Super-users. Addison-Wesley, Reading, Mass., 1985.

Garey, M. and Johnson, D. Computers and Intractability. W. H. Freeman, San Francisco, 1979.

Gevarter, William B. Intelligent Machines: An Introductory Perspective of Artificial Intelligence and Robotics. Prentice-Hall, Englewood Cliffs, NJ, 1985.

Grishman, R. and Hirschman, L. "Question Answering from Natural Language Medical Data Bases". Artificial Intelligence, 11, Nos. 1, 2 (1978):25-43.

Harmon, P. and King, D. EXPERT SYSTEMS: AI in Business. J. Wiley, New York, 1985.

Hinton, Geoffrey E. "Learning in Parallel Networks". Byte, 10, No. 4 (1985):265-273.

Holland, J. Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, Michigan, 1975.

Huang, K., Ghosh, J., and Chowkwanyum, R. "Computer Architectures for Artificial Intelligence Processing". Computer, 20, No. 1 (1987):19-27.

Kelley, A. and Pohl, I. An Introduction to Programming in C. Benjamin/Cummings Publishing Company, Menlo Park, CA, 1984.

Kochan, S. UNIX Shell Programming. Hayden Book Company, Hasbrouck Heights, NJ, 1986.

Lenat, D. "An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search". In Knowledge-Base Systems in AI, ed. L. Lenat, pp. 229-491. McGraw-Hill, New York, 1982a.

Lenat, D. "The Nature of Heuristics". Artificial Intelligence, 19, No. 2 (1982b):189-249.

Meijer, A. and Peeters, P. Computer Network Architectures. Computer Science Press, Rockville, Md., 1983.

Michalski, Ryszard S., Carbonell, Jaime G., and Mitchell, Tom M. Machine Learning. Tioga Publishing Company, Palo Alto, CA, 1983.

Michie, D. (ed.). Introductory Readings in Expert Systems. Gordon and Breach, New York, 1982.

Minsky, M. and Papert, S. Perceptrons. MIT Press, Cambridge, Mass., 1969.

Mitchell, T. "Generalization as Search". Artificial Intelligence, 18, No. 2 (1982):203-226.

Naylor, C. Build Your Own Expert System: Halsted Press, New York, 1985.

Newstead, M. A. and Pettipher, R. "Knowledge Acquisition for Expert Systems". Electrical Communication, 60, No. 2 (1986):115-121.

Nilsson, Nils J. Learning Machines. McGraw-Hill, Inc., New York, 1965.

Nilsson, Nils J. Principles of Artificial Intelligence. Tioga Publishing Company, Palo Alto, CA, 1980.

O'Bannon, R. Michael. "An Intelligent Aid to Assist Knowledge Engineers with Interviewing Experts". In IEEE Western Conference on Expert Systems, pp. 31-36. Computer Society Press of the IEEE, Washington, D. C., 1987.

Rada, R. "Automating Knowledge Acquisition". In Expert Systems: Principles and Case Studies, ed. R. Forsyth, Chapman and Hall, New York, 1984.

Ramamoorthy, C. V., Shekhar, S., and Garg, V. "Software Development Support for AI Programs". Computer, 20, No. 1 (1987):30-40.

Ritchie, G. D. and Hanna, F. K. "AM: A Case Study in AI Methodology". Artificial Intelligence, 23, No. 3 (1984):249-268.

Rochkind, Marc J. Advanced UNIX Programming. Prentice-Hall Inc., Englewood Cliffs, NJ, 1985.

Rosenblatt, F. Principles of Neurodynamics. Spartan Books, New York, 1962.

Simon, Herbert. "Why Should Machines Learn". In Machine Learning, ed. T. Mitchell, pp. 25-38. Tioga Press, Palo Alto, CA, 1983.

Sklansky, J. and Wassel, G. Pattern Classifiers and Trainable Machines. Springer-Verlag, New York, 1981.

Smith, Stephen F. "Adaptive Learning Systems". In Expert Systems: Principles and Case Studies, ed. R. Forsyth, Chapman and Hall, New York, 1984.

Sobell, Mark G. A Practical Guide to UNIX System V. Benjamin/Cummings Publishing Company, Menlo Park, CA, 1985.

Sunshine, Carl A. (ed.). Communication Protocol Modeling. Artech House, Dedham, Mass., 1981.

Utgoff, Paul E. Machine Learning of Inductive Bias. Kluwer Academic Publishers, Boston, 1986.

Waite, M., Martin, D., and Prata, S. UNIX primer Plus. H. W. Sams, Indianapolis, Ind., 1983.

Waterman, D. A. "Generalization Learning Techniques for Automating the Learning of Heuristics". Artificial Intelligence, 1, Nos. 1, 2 (1970):121-170.

Weiss, S. and Kulikowski, C. A Practical Guide to Designing Expert Systems. Rowman & Allanheld, Totowa, NJ, 1984.

Winstom, Patrick. Artificial Intelligence. Second edition. Addison-Wesley, Reading, Mass., 1984.

ACKNOWLEDGEMENTS

For the encouragement, guidance, suggestions during the course of investigation and in preparation of this dissertation, I sincerely thank my major professor, Dr. T. A. Smay.

To the other members of my graduate committee, Dr. D. Grosvenor in the Department of Computer Science, Dr. Way Kuo in the Department of Industrial Engineering, Dr. A. V. Pohm, Dr. C. T. Wright, and Dr. C. S. Comstock in the Department of Electrical Engineering and Computer Engineering, appreciation is extended for their favors granted.

APPENDIX: BASIC PROGRAM MODULES

Part of the basic programs and functions are listed. Most parts of the programs are written in C language, except that the system calls used in the program for the transportation manager are special utilities from UNIX BSD 4.3, and the user interface program is written in Unix C Shell.

The rest of the program can be accessed by contacting the author in the Department of Electrical Engineering and Computer Engineering, Iowa State University.

Structure and Variable Declarations

The database and the knowledge base are declared in frame-like structures. The database is used to save the command syntax and related descriptions, and the knowledge base is used to save statistical information about the implementation of a service in different systems. Other defined constants and global variables are also included in this program head.

```
/## heuristic_struc  #######################################/
#define GKBSIZE   100 /#Size of the global knowledge base #/
#define LKBSIZE    50 /#Size of the local knowledge base  #/
#define SQSIZE     20 /# Max steps in a command sequence  #/
#define HEURISIZE   5 /# Max number of suggestions        #/
#define GKBPATH     "/grad+guest/guest/wuyc/gkb"
                      /# Memory address for the global
                         knowledge base.                  #/

#define SYSDIR      "/grad+guest/guest/wuyc/sys"
                      /# Memory address for the file
                         containing system names of all
                         the systems in GKB.              #/

#define  MAXFAIL    4 /# Max number of learning chances
                         before giving up.                #/

#define NOSYS       3 /# Total Number of systems in GKB   #/

/# Information of suggested command sequences for a
   particular service. #/
struct heuristic
{
  int h_no;                 /# Heuristic suggestion ID #/
  int weight;               /# Weight of the suggestion #/
  int steps;  /# Number of steps in the command sequence  #/
  char  sequence[SQSIZE]; /# Array of command sequences   #/
};

struct kb                  /# A frame in the knowledge base  #/
{
  int    serv_no;        /# Service ID                    #/
  char   serv_name[80]; /# Service description            #/
  float  single; /# Number of systems that have completed
                    the service in a single command.      #/
  float  multi;  /# Number of systems that have completed
                    the service in a multi step command
                    sequence.                             #/
  int    no_heuristic;  /# Number of suggestions          #/
  struct heuristic h[HEURISIZE]; /# Heuristic suggestions #/
};

struct kb2
{
  int no_fail;   /# Failure counter  #/
  int tried_h;   /# Number of used heuristic suggestions  #/
};

struct cmd       /# Database for command information      #/
{
  char filename[20]; /# File name of the command file #/
  char descp[80];    /# Service description          #/
  int  argc;         /# Number of arguments          #/
  char argv[120];    /# Argument descriptions        #/
  int  step;         /# Number of steps              #/
  char syntax[120];  /# Command syntax or sequence   #/
 };

struct kb gkb[GKBSIZE], /# Global knowledge base          #/
          lkb[LKBSIZE]; /# Local knowledge base           #/

struct kb2 lkb2[LKBSIZE]; /# Secondary LKB                #/

struct cmd db[LKBSIZE], /# Single command database        #/
           fc[LKBSIZE]; /# Command sequence database      #/

/# Implementation status variables #/
int new;
int complete;
int change;

int no_services;  /# Number of services in GKB             #/
int no_incomplete;/# Number of incomplete services in LKB #/
int no_unique;    /# Number of special services in LKB    #/
```

Key Functions in the Heuristic Center

Part of the functions used to implement the heuristic center are listed as follows.

```
#include <stdio.h>
#include "heuristic_struc"

char comd[LKBSIZE][80]; /# Table of existed command. #/
int  no_cmd;       /# Number of services in the local
                      service list.#/
int s_no[LKBSIZE];      /# Service ID. #/

/###########################################################/
match(k)
/# Compare the input command with commands in the command
   table. If it is a new command, append it to the command
   table, otherwise, return the service ID corresponding
   to that existed command. #/

int k;  /# The input command is in comd[k]. #/
{
  int i;
  for ( i=0; i <= LKBSIZE; ++i)
   {
    if (k == i) continue;
    if (strncmp(comd[k],comd[i],strlen(comd[k])) == 0)

    /# It is a existed command. The corresponding
       service ID is returned. #/
      return (i);

   }

  /# It is found to be a new command. #/
  return (-1);
}

/###########################################################/
p_step(servno)
/# Calculate the probability of implementation
   in one step. #/

int servno; /# Service ID #/
{
  float i;

  i = gkb[servno].single/
      (gkb[servno].single+gkb[servno].multi);
  printf("\nTHE PROBABILITY IS %f FOR A SINGLE STEP
  SERVICE\n",i);

  return;
}
```

```
/##################################################################/
p_heuristic(servno)
/# Display the heuristic suggestions for the possible
   command sequences. #/

int servno; /# Service ID #/
{
  int i,j,k,c[5];
  char *sq,s[5];
  int a[5],d[5],temp,n;

  /# Find the entry of lkb by its service number. #/
  n = find_entry(servno);

  /# Sort heuristic suggestions by their weight. #/
  for (i = lkb2[n].tried_h; i <= lkb[n].no_heuristic-1; ++i)
  {
    a[i] = lkb[n].h[i].weight;
    d[i] = i;
  }
  for (i =lkb2[n].tried_h; i < lkb[n].no_heuristic-1; ++i)
    for (j = lkb[n].no_heuristic-1; i<j; --j)
      if (a[j-1] < a[j])
      {
       temp = a[j-1];
       a[j-1] = a[j];
       a[j] = temp;
       temp = d[j-1];
       d[j-1] = d[j];
       d[j] = temp;
      }

 /# Display the suggested command sequences. #/
 for (j =lkb2[n].tried_h; j <= lkb[n].no_heuristic-1; ++j)
 {
   printf("\nSUGGESTED COMMAND SEQUENCE %d:\n",j);
   sq = lkb[n].h[d[j]].sequence;
   for (k =0; k <= lkb[n].h[d[j]].steps-1; ++k)
   {
     /# Decompose the filename into sequence numbers. #/
     sscanf(sq,"%[¢_]",s);
     sq = sq + strlen(s) +1;
     c[k] = atoi(s);
     for (i = 0; i <=no_cmd; ++i)
       if ( c[k] == s_no[i])
     break;
     printf("\nFUNCTION : %s\n",gkb[c[k]].serv_name);
     printf("SYNTAX    : %s\n",comd[i]);
   }
 }
```

```
  return;
 }

/##############################################################/
 find_entry(servno)
/# Find the entry to the local knowledge base with the
   specified service ID. #/

 int servno; /# Service ID #/
 {
  int j, n;

  for ( j = 0; j <= no_incomplete -1; ++j)
    if ( lkb[j].serv_no == servno)
      n = j;
      /# Corresponding entry to lkb for the servno is n.#/
  return (n);
 }

/##############################################################/
load_kb(s,pt)
/# Load the knowledge base from the secondary memory. #/

 struct kb *s;  /# Pointer to a frame of the KB. #/
 FILE      *pt; /# Memory address of the KB. #/
 {
   int i,j;

   fscanf(pt,"%[¢\n]",s->serv_name);
   fscanf(pt,"\n%f%f%d\n",&(s->single),&(s->multi),
          &(s->no_heuristic));
   for ( j = 0; j <= s->no_heuristic -1; ++j)
   {
     fscanf(pt,"%d%d%d\n",&(s->h[j].h_no),&(s->h[j].weight),
            &(s->h[j].steps));
     fscanf(pt,"%s",s->h[j].sequence);
     fscanf(pt,"\n");
   }

 }

/##############################################################/
save_kb(s,pt)
/# Save the knowledge base back to the secondary memory. #/

 struct kb *s;
 FILE      *pt;
 {
   int i,j,k;
```

```
      fprintf(pt,"%d\n",s->serv_no);
      fprintf(pt,"%s",s->serv_name);
      fprintf(pt,"\n%f %f %d\n",s->single,s->multi,
      s->no_heuristic);
      for ( j = 0; j <= s->no_heuristic -1; ++j)
      {
        fprintf(pt,"%d %d %d\n",s->h[j].h_no,s->h[j].weight,
        s->h[j].steps);
        fprintf(pt,"%s",s->h[j].sequence);
        fprintf(pt,"\n");
      }

    return;
  }

/########################################################/
assign_kb(s,servno,servname,sg,m,nohu,stp,sq)
/# Initialize the information for a new service. #/

 struct kb *s;      /# Pointer to a frame of the KB. #/
 int servno;        /# Service ID #/
 char servname[80];/# Description of the service. #/
 float sg;          /# If it is a single command service,
                       1 is assigned to sg. #/
 float m;           /# If it is a multi command service,
                       1 is assigned to m. #/
 int nohu;          /# If there is a heuristic suggestion,
                       1 is assigned to nohu. #/
 int stp;           /# The number of steps in the suggested
                       command sequence. #/
 char sq[20];       /# Command sequence #/
 {
   s->serv_no = servno;
   strcpy(s->serv_name,servname);
   s->single    = sg;
   s->multi     = m;
   s->no_heuristic = nohu;
   s->h[0].h_no = 0;
   s->h[0].weight = 1;
   s->h[0].steps  = stp;
   strcpy(s->h[0].sequence,sq);

   return;

 }
```

```
/##########################################################/
update_kb(s,dt_sg,dt_m,stp,sq)
/# Update the information in the knowledge base. #/

 struct kb *s;
 int dt_sg;    /# If it is a single step service,
                  1 is assigned as the increment. #/
 int dt_m;     /# If it is a multi step service,
                  1 is assigned as the increment. #/
 int stp;      /# The number of steps in the suggested
                  command sequence. #/
 char sq[20];
 {
   int i;

   s->single = s->single + dt_sg;
   s->multi  = s->multi  + dt_m;
   if ( dt_sg > 0)
     return; /# It is a single step service. #/

   /# If any command in sequence is "unique",
      stop the knowledge generalization.#/
   for ( i = 0; i <= strlen(sq)-1; ++i)
     if ( sq[i] == '-' ) return;

   /# Update command sequence information. #/
   for (i = 0; i <= s->no_heuristic -1; ++i)
     if (strcmp(s->h[i].sequence,sq) == 0)
     {
       /# Old heuristic succeeded again. #/
       s->h[i].weight++; /# Evaluation function using
                            the most successful criteria. #/
       return;
     }

   /# New heuristic added. #/
   s->no_heuristic++;
   s->h[s->no_heuristic-1].h_no = s->no_heuristic -1;
   s->h[s->no_heuristic-1].weight = 1;
   s->h[s->no_heuristic-1].steps  = stp;
   strcpy(s->h[s->no_heuristic-1].sequence,sq);

   return;
 }
```

```
/###############################################################/
update_new(sys,servno)
/# Set the corresponding bit in the status variable "NEW"
   of each system that is not under training. This function
   is called when a new service is added to the GKB. #/

 int servno;       /# Service ID #/
 char sys[5][80]; /# Systems that are not under training. #/
 {
   FILE *pt, *update_pt;
   int new, complete, change; /# Implementation status
                                 variables. #/
   int p; /# Bit mask #/
   int k;

   p = 1;
   for ( k = 0; k <= NOSYS-2; ++k) /# Total number of
                          systems in GKB is NOSYS. #/
   {
   update_pt = fopen(sys+k,"r");
   fscanf(update_pt,"%d%d%d",&new,&complete,&change);
   fclose(update_pt);
   new = new|(p << servno);  /# Set the corresponding bit
                                in the variable "NEW". #/
   update_pt = fopen(sys+k,"w");
   fprintf(update_pt,"%d %d %d",new,complete,change);
   fclose(update_pt);
   }
 }

/###############################################################/
update_change(sys,servno)
/# Set the corresponding bit in the status variable
   "CHANGE" of each system that is not under training.
   This function is called when the information of a service
   in the GKB is changed. #/

 int servno;
 char sys[5][80];

 {
   FILE *pt,*update_pt;
   int new, complete, change;
   int p;
   int k;

   p =1;
   for ( k = 0; k <= NOSYS-2; ++k)
   {
   update_pt = fopen(sys+k,"r");
```

```
    fscanf(update_pt,"%d%d%d",&new,&complete,&change);
    fclose(update_pt);
    change = change|(p << servno); /# Set the corresponding
                                      bit in "CHANGE". #/
    update_pt = fopen(sys+k,"w");
    fprintf(update_pt,"%d %d %d",new,complete,change);
    fclose(update_pt);
    }
  }

/###########################################################/
find_other_systems(a)
/#Find the systems that are not currently under training. #/

 char a[5][80];
 {
   FILE *pl, *p2;
   int i;
   char current[80], /#Name of the system under training. #/
        b[80];

   pl = fopen("current","r");
   fscanf(pl,"%s",current);
   fclose(pl);

   p2 = fopen(SYSDIR,"r");  /# All the system names of the
                               systems in GKB are kept in
                               SYSDIR. #/

   i = 0;
   while(fgets(b,80,p2) != NULL)
   {
    if (strncmp(b,current,strlen(current)-1) == 0) continue;
    strncpy(a[i],b,strlen(b)-1);
    strcat(a[i],"/update");
    i++;
   }
   fclose(p2);

   return;
 }
```

Basic Parts in the Transportation Manager

The transmitter and receiver are implemented by using the concept of UNIX processes, which can be generated simultaneously to simulate the concurrent situation.

The READY signal is transmitted from the receiver to the transmitter through the pipe, which is also created by the special UNIX system call "pipe" for interprocess communication.

```
/##############################################################/
/# Create a pipe to link the TRANSMITTER and RECEIVER. #/

if (pipe(pfd) == -1) {
 printf("\npipe can not be created\n");
 exit(1);
}

/##############################################################/
switch(pid=fork()) {
    case 0:                   /# Child process #/
       /# RECEIVER #/
       i = 0;
       j = 0;
       while(read(csout,&d[i],1))  /# Read a byte from the
                                      connected system. #/
       {
        write(1,&d[i],1);
        if ( (d[i] == '?') || (d[i] == '>') ) {
        /# System prompt is received. #/
         if (j <= final) {
          /# Continue transmission #/
            if(strncmp(&d[1],rp[j],strlen(rp[j])-1) == 0 )
            /# Expected response received, send the sequence
               number as the READY signal to TRANSMITTER. #/
              write(pfd[1],&j,1);

             else
             /# Unexpected response received, send error
                signal to TRANSMITTER. #/
                {
                  j = WRONG;
                  write(pfd[1],&j,1);
                }
          j++;
         }
        }
        if (d[i] == '¢M')  {
            write(resbf,d,i+1);
          i = -1;
        }
        i =(i+1) % RBF_SIZE;
       }

/##############################################################/
default:  /# Parent process #/
  /# TRANSMITTER #/
  while(1)
  {
    read(pfd[0],&s_ready,1); /# Wait the READY signal. #/
```

```
     if (s_ready == WRONG)
     /# Error signal is received. #/
       break;

     sleep(1); /# Time is required for next transmission. #/

     if (file_transfer)
       {
        cp_pt = fopen(cpfl,"r");
        while( fgets(b,80,cp_pt) != NULL)
        {
         b[strlen(b) -1] = '¢M';
         write(csout,b,strlen(b));
         sleep(1);
        }
        write(csout,"¢C",1);
       }
    strncpy(b,sp[s_ready],strlen(sp[s_ready])-1);
    b[strlen(sp[s_ready])-1] = '¢M';
    write(csout,b,strlen(sp[s_ready]));
    if (s_ready == final) break;
   }

   kill(pid,9);   /# Terminated the child process #/
   quit();
   return;
}
```

User Interface

This is an application program written in Unix C Shell, in which a service table is displayed to help the user in specifying an object system and selecting operation modes.

```
/# Display the systems available. #/
while (1 == 1)
  echo "     "
  echo "1: Wylbur"
  echo "2: Vms"
  echo "3: Unix"
  echo "4: EXIT"
  echo 'INPUT SYSTEM ID>'
  set sys = ¢head -1¢

  switch  ($sys)
  /# Move to the specified directory according
    to the specified system. #/
    case 1:
      cd wyldir
      breaksw
    case 2:
      cd vaxdir
      breaksw
    case 3:
      cd unixdir
      breaksw
    case 4:
      break;
  endsw

  while ( 1 == 1)
    /# Display the operation modes. #/
    echo "     "
    echo "1: Training Mode"
    echo "2: Using Mode"
    echo "3: EXIT"
    echo "INPUT MODE NUMBER>"
    set mode = ¢head -1¢
    switch ($mode)
      case 1:
        setup     /# Training mode #/
        breaksw
      case 2:
        use2 $sys /# Using mode #/
        breaksw
      case 3:
        break
    endsw
  end
  cd ..
end
```