AD-A218 281

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AN INTELLIGENT COMPUTER-ASSISTED
INSTRUCTION SYSTEM FOR
UNDERWAY REPLENISHMENT

by

Patricio José Salgado-Zapata

June 1989

Thesis Advisor   Neil C. Rowe

90 02 15 024

REPORT DOCUMENTATION PAGE

1a Report Security Classification Unclassified		1b Restrictive Markings		
2a Security Classification Authority		3 Distribution Availability of Report		
2b Declassification Downgrading Schedule		Approved for public release; distribution is unlimited.		
4 Performing Organization Report Number(s)		5 Monitoring Organization Report Number(s)		
6a Name of Performing Organization Naval Postgraduate School	6b Office Symbol (if applicable) 33	7a Name of Monitoring Organization Naval Postgraduate School		
6c Address (city, state, and ZIP code) Monterey, CA 93943-5000		7b Address (city, state, and ZIP code) Monterey, CA 93943-5000		
8a Name of Funding Sponsoring Organization	8b Office Symbol (if applicable)	9 Procurement Instrument Identification Number		
8c Address (city, state, and ZIP code)		10 Source of Funding Numbers		

	Program Element No	Project No	Task No	Work Unit Accession No

11 Title (include security classification) AN INTELLIGENT COMPUTER-ASSISTED INSTRUCTION SYSTEM FOR UNDERWAY REPLENISHMENT

12 Personal Author(s) Patricio José Salgado-Zapata

13a Type of Report Master's Thesis	13b Time Covered From To	14 Date of Report (year, month, day) June 1989	15 Page Count 76

16 Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes			18 Subject Terms (continue on reverse if necessary and identify by block number)
Field	Group	Subgroup	Intelligent Computer-Assisted Instruction (ICAI), UNREP.

19 Abstract (continue on reverse if necessary and identify by block number)

This research discusses the design, implementation, and testing of the UNREP system, an Intelligent Computer-Assisted Instruction (ICAI) tutoring system to simulate Underway Replenishment operations by training two students simultaneously on separate computer workstations. Each student plays the role of the Officer of the Deck (OOD) aboard each of the ships involved. Emergency situations are included to add realism to the simulation.

While several different ICAI systems have been developed in the past, few have focused on the coordination aspects of applications which involve cooperation in joint activities, such as military operations. Artificial Intelligence (AI) techniques and programming tools were employed to construct this system. Education and training in the military, ICAI systems for military applications, ICAI general characteristics, knowledge representation, and time and task coordination are some of the topics discussed in this thesis.

20 Distribution Availability of Abstract ☒ unclassified unlimited ☐ same as report ☐ DTIC users	21 Abstract Security Classification Unclassified		
22a Name of Responsible Individual Neil C. Rowe	22b Telephone (include Area code) (408) 646-2462	22c Office Symbol 52Rp	

DD FORM 1473,84 MAR   83 APR edition may be used until exhausted   security classification of this page
All other editions are obsolete

Unclassified

An Intelligent Computer-Assisted
Instruction System for
Underway Replenishment

by

Patricio José Salgado-Zapata
Lieutenant Junior Grade, Ecuadorian Navy
B.S., United States Naval Academy, 1985

Submitted in partial fulfillment of the
requirements for the degree of
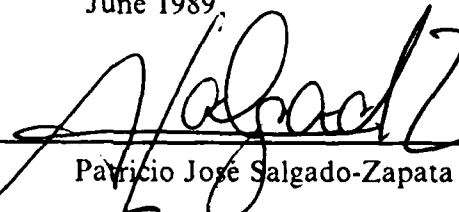
MASTER OF SCIENCE IN ENGINEERING SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1989
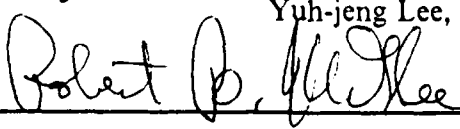
Author: _____

Patricio José Salgado-Zapata

Approved by: _____

Neil C. Rowe, Thesis Advisor

_____

Yuh-jeng Lee, Second Reader

_____

Robert B. McGhee, Chairman,
Department of Computer Science

_____

Kneale T. Marshall,
Dean of Information and Policy Sciences

ii

ABSTRACT

This research discusses the design, implementation, and testing of the UNREP system, an Intelligent Computer-Assisted Instruction (ICAI) tutoring system to simulate Underway Replenishment operations by training two students simultaneously on separate computer workstations. Each student plays the role of the Officer of the Deck (OOD) aboard each of the ships involved. Emergency situations are included to add realism to the simulation.

While several different ICAI systems have been developed in the past, few have focused on the coordination aspects of applications which involve cooperation in joint activities, such as military operations. Artificial Intelligence (AI) techniques and programming tools were employed to construct this system. Education and training in the military, ICAI systems for military applications, ICAI general characteristics, knowledge representation, and time and task coordination are some of the topics discussed in this thesis.

iii

THESIS DISCLAIMER

The reader is cautioned that computer programs developed in this research may not have been exercised for all cases of interest. While every effort has been made, within the time available, to ensure that the programs are free of computational and logic errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my thesis advisor, Professor Neil C. Rowe, who provided guidance and assistance throughout this thesis research.

Writing a thesis is not just an intellectual endeavor, it is a time consuming and emotional experience; with the support and encouragement of my beautiful wife Maria Elena, this task has been easier to undertake. I would also like to thank my parents, although far away, their love and dedication is a key to what I have become. Finally, but not least, thank you Carlos Andrés, though being too little to understand, you have brought even more happiness into this exciting and exhausting episode of my life.

I. INTRODUCTION

During peacetime the top goal of any military organization is to maintain its personnel and equipment ready to go into action whenever needed. The best way to achieve personnel readiness is through continuous education and training. In this effort almost 20 billion dollars were spent in 1988 by the US Defense Department in formal training, and almost 200,000 military and civilian personnel were required to provide instruction [Ref. 1]. With today's technological and economical complexities, the need for expertise has never been greater. The increasing number of military systems together with their technological sophistication influences the military to look for more personnel to master complicated and varied subject material and skills. The percentage of high-skill personnel required by the US Navy increased from 23 percent in 1945 to 42 percent in 1980 [Ref. 2].

Education in the military is becoming a problem nowadays for three major reasons. First, teaching and training personnel in real world situations can result in many unnecessary expenses; e.g., damage or loss of expensive equipment, injuries of personnel, expenditure of fuel or other energy sources, and loss of valuable time. Second, the number of military systems is increasing while the number of military personnel is decreasing in some countries; the military cannot afford to assign already available high-skilled personnel as instructors or tutors, for that would make the situation worse than it is. Third, military units like ships are usually deployed and constitute autonomous units; their access to qualified instructors and instructional facilities is limited, if not impossible.

New approaches to military training and instruction are necessary if the goal of complete personnel readiness is to be achieved. If technology advances and

1

sophistication are invading the modern world, why not take advantage of them in the area of education. Computers have become a widely used and easy-to-access commodity during the last decade, so it is not surprising that the idea of using computers for instructional purposes has already been introduced. Computer-Based Instruction has been used for several years now for different applications. The term Computer-Based Instruction, or CBI, refers to all uses of computers in instruction.

Originally CBI took the name of Computer-Assisted Instruction, or CAI. As its name implies, this approach was designed to aid students in the process of learning, but it did not have a great impact primarily because the majority of these systems just printed prepared text and ran prestored drill-and-practice sessions [Ref. 3 pg 225]. CAI systems do not usually adapt well to the students' needs and speed of learning. New advances in the area of Artificial Intelligence (AI) are giving new hope to this approach to instruction. Computers can now be programmed to simulate a tutoring process similar to the human counterpart. This new method is Intelligent Computer-Assisted Instruction, or ICAI. An ICAI system is able to understand complex feedback from the student and modify its tutoring strategy according to the student's level of learning.

The purpose of this thesis is to examine the feasibility of an ICAI system that tutors two or more students at the same time and on the same application. Underway Replenishment was chosen as the application domain since it is a military procedure which requires the step-by-step interaction of two ships simultaneously in order to achieve its goal.

A. UNREP

The primary mission of a navy is to maintain its country's sea routes of supply open during peace and war times. The most efficient way to accomplish this mission is to keep warships always at sea, well stocked and ready to go into action whenever necessary. Underway Replenishment, also known as Replenishment at Sea, or simply

UNREP, enables naval units to extend their operational time at sea. Fuel, supplies, ammunition, and other types of cargo including personnel, are transferred from large ships to less self-sufficient ships while cruising side by side in the open seas, making it possible for a ship to remain at sea up to an entire period (theoretically) between major overhauls or drydocking.

Useful and practical as this military concept seems, it is a complex procedure, requiring special knowledge and expertise from the officers in charge of executing the maneuver. During replenishment operations the danger of collision is much greater than under normal circumstances. The unusual proximity of the ships involved and the complex arrangement of lines and hoses connected between them reduces the maneuverability and increases the vulnerability to attacks during war time. Open storage compartments, exposed material, and occasionally dangerous or explosive cargo are some of the hazards to which personnel involved are exposed.

B. SCOPE OF THESIS

The UNREP ICAI system (referred to as the UNREP tutor) described in this thesis is by no means a complete course in UNREP operations. The UNREP tutor has been designed to supplement the courses and training offered by a navy on the subject matter, not replace them. The tutor focuses its instruction on orders issued by the Officers of the Deck (OOD's) aboard each UNREP ship, since they are the ones that monitor this procedure step by step. Potential users of this system could be Midshipmen or Junior Officers in the navy who are training to get the OOD qualification.

The UNREP tutor was implemented using artificial-intelligence techniques in the Prolog programming language.

C. PREVIEW

Chapter II provides background on Intelligent Computer-Assisted Instruction (ICAI) and its components, and describes ICAI systems sponsored or developed by the

military, including the application-independent METUTOR program. Chapter III provides background on Underway Replenishment (UNREP) and describes its procedure from the Officer's of the Deck point of view. Chapter IV discusses the actual design and implementation of the UNREP tutor. Chapter V summarizes the performance of the system, addressing issues such as memory requirements, time considerations, accuracy, and complexity. Chapter VI discusses the major achievements and limitations of the system, providing recommendations for future system enhancements. Appendices A and B contain test runs. Appendices C and D contain the Prolog code for the knowledge representations of the UNREP system. Appendix E contains the Prolog source code developed by Professor Rowe.

II. ICAI SYSTEMS IN MILITARY APPLICATIONS

Many ICAI systems have been designed and implemented. Some of them have been tested and evaluated, proving to be successful and efficient under real training situations. Others, on the other hand, are far away from practical. This chapter gives an overview of Intelligent Computer-Assisted Instruction, and briefly describes military ICAI systems, pointing out features of interest.

A. OVERVIEW OF ICAI SYSTEMS

Intelligent Computer-Assisted Instruction, or ICAI, is the area of computer science that deals with the design, development, and implementation of computer-based systems that provide adaptive and dynamic instructional environments to a user, applying artificial-intelligence techniques [Ref. 4: p. 15].

Computer-based instruction, or CBI, has not always been considered a field of interest specific to computer science. CBI was first explored by educational psychologists under the name of Computer-Assisted Instruction, or CAI. CAI systems on their early years were developed primarily as supplemental tools to the traditional methods of instruction. The approach of CAI was that of using computers as an efficient method of translating a teacher's pedagogical decision into a program [Ref. 5], like a computer textbook, without checking the student's real understanding of the subject. The human teacher was always taking care of the tutoring aspects, and the computer was merely a tool.

Later on, ICAI research was introduced into the computer-science field, and a more ambitious task was undertaken. The interest of ICAI did not just comprise instruction and learning aspects, but also the development of computer systems that would "effectively capture human being's learning and teaching processes" [Ref. 4: p. 38]. "Adaptive"

and "dynamic" are key words in the definition of ICAI. The former refers to the ability of the computer to adapt to the needs of the student depending on his/her level of understanding: some students will grasp a point quickly, while others will need a different approach to learn the same subject. In order for a system to be adaptive, it must also be dynamic. Through interactions with the student during the tutoring session, the ICAI system could determine to what level the student is assimilating the skill or subject being taught. The system could not only get feedback on how many questions have been answered correctly by the student, but evaluate these answers and try to define the problem area of the student. Based on this evaluation, the system could change its teaching style, speed, and depth.

Artificial-Intelligence (AI) techniques are indispensable in the design of ICAI systems. Natural-language understanding and representation for input and output is a necessary feature for a more efficient interaction between the student and the system. Knowledge representation is also required in an ICAI system in order to encapsulate all the expertise on the subject being taught. Methods of inference provide the system the ability to generate problems for the student. Some applications require algebraic simplification, symbolic integrations, theorem proving, etc. AI techniques and methods make these tasks possible in an efficient way [Ref. 3: p. 227].

Although there are no definite and precise characteristics which specify whether a tutoring system is an ICAI system or not, four components are responsible for providing the basic features of most existing ICAI systems. These components are the expertise module, the tutorial module, the student model, and the inference engine [Ref. 3: pp. 229-235]. They represent respectively the subject to be taught, the tutoring strategy, the theorized model of the student's knowledge, and the method to determine how much the student has learned so far. Some ICAI systems contain well-defined and separate

modules, while others encapsulate the functions of the four distinguishing components into one single module.

B. MILITARY ICAI SYSTEMS

This section will discuss ICAI systems that have been, or are currently being developed by the military and for military training applications. Fletcher [Ref. 6] has gathered a list of ICAI systems which meet our criteria. We briefly describe seven of those systems plus another system that was developed at the Naval Postgraduate School and constitutes the backbone of this thesis.

The Sophisticated Instructional Environment (SOPHIE) ICAI system is one of the first and most important contributions, not only to the military training research community, but to the ICAI field in general. SOPHIE teaches problem-solving skills by having the student take measurements on a simulated electronic piece of equipment which has a malfunction. The student's goal is to find the fault by troubleshooting the simulated circuit [Ref. 3: p. 297]. SOPHIE's contribution to the ICAI field is the introduction of device-based simulation to support checking of student inferences, as well as heuristic strategies to allow question generation and answering mechanisms. [Ref. 6 and Ref. 7: pp. 227-281]. STEAMER is another ICAI system that was developed in the late 1970's. This system is based in the simulation of a ship's steam propulsion plant. STEAMER's main goal is to train operators by helping them understand this complex application through interactive graphical interfaces [Ref. 4 : pp. 114-115]. Although the actual implementation of reasoning mechanisms in the STEAMER project is purely mathematical, its underlying principles have inspired research in the areas of mental models and abstraction simulation in AI [Ref. 8: pp. 79-88].

QUEST (Qualitative Understanding of Electrical System Troubleshooting) is similar to the SOPHIE system in that the goal of the system is to provide the student with a learning environment so that he/she can solve circuit problems. QUEST, however, relies

on graphic simulation as well as casual explanations of circuit behavior to support the student's learning process [Ref. 8: pp. 88-89]. Like STEAMER, QUEST investigated the research area of mental models, but it also emphasized on the area of qualitative reasoning.

The Intelligent Maintenance Training System (IMTS) is intended to simulate the functions of different devices and train students in their maintenance, therefore acting as an operational maintenance trainer. The system's current application is the simulation of the SH-3 helicopter's blade-folding mechanism. The main characteristic of IMTS is its emphasis on the interface with human instructors, allowing them to control some of the operating modes of IMTS. This system builds a conceptual model of the student's skills and measures his general knowledge and learning preferences in order to select its tutoring strategy and type of problems. MACH-III is the Maintenance Aid Computer for HAWK - Intelligent Institutional Instructor. It provides training in maintenance of electronics and electromechanical systems in general; currently the subject domain is the maintenance of the high-powered illumination radar of the HAWK air defense system. The MACH-III system supports three modes of instruction: demonstration, step-by-step guided practice, and free-form monitored practice.

TRIO, a Trainer for Radar Intercept Operations, includes real-time simulation, speech synthesis, and speech recognition capabilities in the training of F-14 interceptor pilots and radar officers. The students solution to an air-intercept problem is compared to the solution generated by an expert knowledge base. TRIO is not only concerned with the tutoring of problem-solving methods, but it also trains students on how to react fast enough when confronted with radar warnings. Specific function-modules generate warnings to the student while monitoring his performance. Another system that has a time-constrained application is the Intelligent Conduct of Fire Trainer (INCOFT) system. It is intended to train students in the operation of the engagement control station

8

of the PATRIOT air defense system. INCOFT uses basically the same approach as TRIO, with the exception of the speech capabilities since a spoken interaction is not that important in the INCOFT application as it is in the TRIO.

1. METUTOR

METUTOR is a general-purpose Means-Ends Tutor developed by Professor Rowe at the Naval Postgraduate School. Three major applications have been adapted to the system so far: the MEFIRE, the CPR, and the Network-Mail tutors. The first trains students on fire-fighting procedures aboard a Navy ship; the second uses the medical field of CPR as its subject-domain [Ref. 9]; the third teaches students to use the Arpanet MM mail facilities [Ref. 10]. The METUTOR system is written using the Prolog language and a version of it is contained in appendix D.

The expertise module of an ICAI system is usually divided into a knowledge base and domain-reasoning methods. In the METUTOR system, the knowledge base is kept separate from the rest of the system since the tutor is intended to be used for teaching or training various subject domains. The knowledge base used in this thesis research is part of the discussion of chapter 4. The domain-reasoning methods in the METUTOR use a means-ends analysis program[1] to reason top-down from abstract goals. The top level is a recursive **means-ends** predicate which has four arguments: **State, Goal, Oplist, and Goalstate.** The **State** is a complete list of facts which are true in a state. The **Goal** is a list of facts that are required to be true in the goal state. The **Oplist** is a list of operators required to reach the goal state from the starting state. The **Goalstate** is a complete list of true facts at the goal state [Ref. 11].

The tutorial module in the METUTOR encapsulates tutoring strategies and monitors the student model accordingly, guiding the dialogue between the system and the student. Two general approaches are used by the METUTOR system. The first one

---

[1] This means-ends program is included in the METUTOR11 listing of appendix D.

provides immediate corrective responses to the student's actions, while the second one temporarily allows the student to follow an incorrect path of actions commenting on the error only after the student has realized his own mistake. The tutoring rules used in the METUTOR system are handled by the predicate **handle_student_op**.[2]

The METUTOR's student model uses a stack representation of the student's applied operators in order to determine the student's level of understanding. This stack is compared to the expert's solution path of operators by the inference-engine module, and the tutoring strategy to be used is determined by figuring out what the student is actually trying to do at each step during the tutoring session. The following tutoring replies are used by the tutorial module in the METUTOR program:

- **OK.** The student's response matches that of the expert module means-ends analysis.

- **The possible operators are: < list of operators >.** The student requested a list of valid operators by entering a help word.

- **I assume you mean < operator >.** The student made a spelling error when entering the chosen operator. The tutor recognizes the operator though.

- **Not a valid operator--please choose one of: < possible operators >.** The student entered an invalid operator, or a string of nonsense characters, or made too many spelling errors. He/she has to enter an operator again.

- **That operator requires that < required precondition list >.** The facts in the precondition list must first be achieved in order to apply the operator.

- **That will not affect anything.** The student's chosen operator does not have any effect on the current state.

- **You cannot ever succeed if you do that.** The operator chosen by the student would create a state from where it would be impossible to reach the goal state. The student has to type another choice of operator.

- **That does not seem immediately helpful, but I will try it.** The student is allowed to follow his/her own solution path. This tutoring strategy does not tutor immediately, but sets up a flag in the database, indicating that the student seems to be pursuing a digression.

- **I will try it but it is not recommended first when < recommended operators >.** The student has picked an operator which does help reach the goal state, but it is not the highest-priority operator.

---

2 This predicate is also contained in the METUTOR11 listing of appendix D.

- **Not the operator I would choose, but let us try it anyway.** The tutor cannot really figure out what the student is trying to do, but allows him/her to apply it anyway.

- **You are returning to a previous state.** The student is returning from a digression. Works in conjunction with the "That does not seem immediately helpful, but I will try it" strategy.

- **Do you see now that your first choice of action in the state with the facts < previous state description > was not the best choice; the < operator > action would have been better.** After allowing the student to continue with his/her solution path, the tutor hopes the student realized why his/her choice of operator was not a high priority one.

The inference engine module of the METUTOR is mixed in with the means-ends tutor code. The means-ends tutor works in parallel with the student by asking an input operator before every action. The inference engine checks if the student's choice of action agrees with the tutor's internal recommendation; if it does, the tutor immediately applies the student's operator; otherwise, the inference engine tries to figure out what the student is doing by means of a complex analysis of the hypothetical state created by the student. The tutorial module then takes over again and tutors the student accordingly.

11

III.   OVERVIEW OF UNDERWAY REPLENISHMENT

Underway Replenishment (UNREP), also known as Replenishment at Sea, is the term applied to the transfer of fuel, munitions, supplies, and men from one vessel to another while ships are underway.   Most of the transfers are performed from special-purpose replenishment ships to combatant ships, and major combatants, like carriers for example, are also capable of refueling smaller ships.   Even the smallest ships can and do exchange light freight, mail, and personnel with other smaller ships.   In any case, the larger ship, or the ship from where the cargo (fuel and personnel are also defined as **cargo** in this case) is transferred from, is called the **delivery ship**, and the smaller ship, or the destination of the cargo, is called the **receiving ship**.

The ability of men to work together smoothly is important in a replenishment operation.   The necessity for adequate training is increased in transfer operations by the fact that crews from different ships and from different nations are called on to work together although they may never have had any contact with one another before.   A high degree of teamwork and coordination must be achieved.   Preparing the cargo to be transferred, rigging (setting up gear for working), and handling gear are all special skills and techniques which are not within the duties of the Officer of the Deck.   But certain aspects of replenishment do concern the Officer of the Deck, also referred as the **OOD**. He should know who is responsible for rigging the special gear required, how long this preparation takes, when to station (assign positions) the special details (small group of personnel temporarily assigned to fulfill a precise requirement, in this case a job related to replenishment at sea), and particularly he should be familiar with the UNREP step-by-step procedure.

The tutoring program of this thesis specifically focuses on the Underway Replenishment *procedure* from the OOD's point of view.

A. PROCEDURES

The UNREP procedures for the U.S. Navy are described in detail in the NWP 14 [Ref. 12]. Replenishment is accomplished with both the delivery and receiving ships steaming side-by-side on parallel courses at a predetermined speed. But first, in order for the ships to be alongside (side-by-side), one of them must approach the other.

1. Making the approach

A typical Underway Replenishment begins when a task-force commander arranges a rendezvous at sea with the group, and then orders the start of the replenishment operation with a given ordered course and speed. The approach is executed as follows. When steady on the ordered course and speed, the delivery ship will fly (display or exhibit) the signal flag **ROMEO at the dip** (signal-flag hoisted about six feet down from the full-up position) **on the rigged side**. She (ships are referred as feminine) will fly **ROMEO close-up** (signal-flag hoisted full-up) when ready to receive, in other words, when the replenishment side has been rigged. The receiving ship which is about 500 yards on the quarter (relative bearing halfway between astern and abeam on either side of delivery ship) will also fly ROMEO at the dip on the rigged side when ready to come alongside. She will fly ROMEO close-up when she is commencing her approach. The receiving ship usually approaches the delivery ship because of the smaller's ship better maneuverability.

Once both ships are alongside (actually separated by a distance of about 100 feet), the delivery ship **shoots the gun-line** (one of the methods to send the first line from the delivery ship to the receiving ship) and the receiving ship **receives and secures the gun line**. As soon as the first line is secured, and the transfer rigs are passed and connected,

13

both ships **haul-down ROMEO** (lower signal flag completely). This step completes the approach phase of the UNREP procedure.

2. Replenishment and station-keeping

During the transfer of flammable or explosive items, such as gasoline, fuel oil, and ammunition, both ships involved **fly BRAVO at the fore**. In order to reduce the probabilities of collision by having both ships maneuver to keep their distance and proper station, only the *receiving ship* is responsible for maneuvering. The receiving ship must ensure that the specified **distance between ships** is maintained during the approach and during the replenishment, as well as keeping her station exactly abeam from the delivery ship. The delivery ship is only responsible for maintaining the predetermined replenishment course and speed.

Keeping station abeam is simplified by watching the distance line and adjusting the course accordingly. The distance line, among the first to be passed across, serves as an indicator for the distance between ships; by watching it, the OOD of the ship will know immediately that his ship is coming in too close or going out too far. Also, by watching a mark in the other ship or observing the angle that the distance line makes with the ship's side, the OOD can determine if the delivery ship is bearing slightly ahead or behind, and get back into station by slowly adjusting the speed.

3. Departure

Fifteen minutes prior to the completion of alongside refueling or replenishment, the receiving ship **hoists PREP at the dip**. Upon completion, the receiving ship **hoists PREP close-up**, meaning that she is starting to disengage from the delivery ship. The receiving ship then slowly increases speed and clears ahead and away from the delivery ship.

4. Emergencies

Several emergency situations can arise during replenishment operations. The prime objective during any emergency is to perform an **emergency breakaway**, in other words, disengage as soon as possible in order to avoid a collision or expose the lines to excessive tension and therefore endanger the involved personnel.

In the event of an emergency, both ships have to make sure that the personnel handling rigs and equipment know about the emergency so they can disengage couplings and other lines with dispatch. The emergency should be announced over the ships' loudspeaker, and the EMERGENCY flag must be flown. As soon as all the lines and couplings have been disconnected, both ships clear and sail away from each other.

B. APPLICATION COVERAGE

Trying to represent a complete real replenishment procedure can lead to an extensive and complicated model. In order to simplify this application, several assumptions have been made.

For purposes of this thesis, the UNREP procedures represented daytime operations. Nightime operations involve other signals and increased difficulties arise, making the model too complicated. Also, it is assumed that the only method of communication between the two ships is by means of signal flags. In reality, communications are crucial for the success of the operation, and that is why several means of communication are used, such as sound-powered telephones, radio, hand signals, visual light signals, and even megaphones.

Numerical data, such as actual speeds, courses, ship bearings, and distances between ships, are not used in the modeling since in reality each ship has different characteristics, so speeds and course adjustments would depend on the type of ship.

Emergencies in an actual UNREP could happen any time and in many different ways. Again, for simplification purposes, only two types of emergencies are simulated.

15

They are steering system failure and excessive line tensioning due to improper maneuvering of the ship.

IV. UNREP: A MULTI-USER ICAI SYSTEM

A. SYSTEM ENVIRONMENT AND OVERVIEW

UNREP is an ICAI system developed and implemented on an Integrated Solutions Inc. (ISI) Optimum V Workstation computer, using the MPROLOG language. MPROLOG (Modular Programming in Logic) is a modular version of the logic programming language PROLOG.

Given that the Underway Replenishment application requires the interaction of two ships, the UNREP system was designed to handle two students simultaneously, each of them acting as the OOD aboard each of the ships involved. The UNREP system requires that each of the two students have access to individual workstations, both of them connected to a common memory base. Two knowledge bases representing the UNREP procedures for the delivery and receiving ships' procedures, respectively, were written and adapted to the application-independent METUTOR11 program.[3] The multi-user approach was accomplished by adding to the METUTOR11 program the ability to handle several students (preferably not more than two as is discussed in chapter VI) at the same time, each of them playing different roles within the same application. The multi-user capabilities of the system do not affect the original single-user tutoring facilities of the METUTOR11; as a matter of fact, a single-user version of the UNREP tutor can be readily obtained by slightly modifying the knowledge bases of the two-user system.

---

[3] Chapter II, section B.1, reviews this program, and appendix E contains a listing of it.

17

B. UNREP KNOWLEDGE BASES

The knowledge bases for the UNREP tutor are contained in the files MEUNREPDEL and MEUNREPREC.[4] They represent the delivery ship and receiving ship Underway-Replenishment procedures respectively.

The first step in the development of the knowledge representations was to gather the subject matter required for the Underway Replenishment procedures. The domain expertise for each of the knowledge bases was extracted from the NWP 14 Underway Replenishment Procedures Manual [Ref. 12]. Although one of the main advantages of ICAI systems in general is the capability to bring together the knowledge of several experts into a single knowledge base, we thought that the information contained in the above mentioned publication was sufficient to meet the objectives of this thesis research.

Next, the expert knowledge was translated into a format that could be understood by the METUTOR11 program. The Underway Replenishment procedure is an ordered sequence of actions, starting from an initial given state, the **starting state**, and ending at a final state called the **goal state**. In order to reach the goal state, a recommended solution path must be followed from the starting state, and going through intermediate states. This recommended path can be achieved by "performing" successive actions, or in other terms, by applying an **operator** at each given state, until the goal state is achieved. A **state** is characterized by a set of **facts**, also considered a list of true facts. Each of the facts in the goal state must be true by the end of a successful tutoring session.

Facts can be achieved (added to the true fact list of a state) by applying recommended operators for them. A **recommended** predicate in the knowledge base summarizes such recommendations [Ref. 11: pp. 268-269]:

---

[4] Appendices C and D contain the source code of the MEUNREPDEL and MEUNREPREC knowledge bases respectively.

recommended([< fact to be achieved >], < operator >).

The **recommended** rules are ordered according to the priority in which the operators must be applied.

An operator can only be applied if the current state contains all facts that have been defined as prerequisites to that operator. The **precondition** predicate serves this purpose:

precondition(< operator > ,[< list of required facts >]).

In this predicate, the second argument is the list of facts that must be present in the current state if the operator is to be applied in that state. The second argument can also be an empty list if there are no prerequisites needed to use that operator.

After an operator is applied, the list of true facts, also known as the **state description**, can change, adding new facts to the state, and deleting others. Two predicates are defined in the knowledge base to handle these changes:

addpostcondition(< operator > ,[< list of facts to be added >]).

deletepostcondition(< operator > ,[< list of facts to be deleted >]).

Again, the second argument lists can be empty lists depending on the operator postconditions.

It is important to mention that the METUTORII tutorial module checks the knowledge base before starting the tutorial session with the student. It makes sure that each operator is defined by the required two-argument predicates (recommended, precondition, addpostcondition, and deletepostcondition predicates); in addition, it checks that a solution of operators exists to achieve the goal state, given a starting state. These tasks are always performed due to the application-independent nature of the METUTORII program.

Other optional predicates may be defined in the knowledge base to add flexibility and enhance the realistic representation of the domain application. Three and

four-argument **addpostcondition** predicates may be defined for an operator. These predicates are:

   addpostcondition(< operator > ,[< list of prerequisite facts >],[< facts to be added >]).

   addpostcondition(< oper. > ,[< prereq. facts >],[< facts to be added >], < message >).

The three-argument predicate adds the facts listed in the third argument only if the additional second argument facts are part of the current state description. The fourth argument is a message that appears on the screen when the list of required facts is true in the state description. The **deletepostcondition** predicate can also be defined using these three and four-argument formats, and they work similarly.

    A **nopref** predicate is used when the priority of two operators in the recommended rules is not of importance; in other words, the order in which the operators may be applied is arbitrary for the tutor. Its format is as follows:

   nopref(< operator 1 > , < operator 2 >).

    A very important optional predicate is the **randsubst** predicate:

   randsubst(< operator > ,[[< deleted fact > , < added fact > , < probability > ,
          < opt. message >],[< another opt. quadruple >],. . .]).

This predicate introduces the factor of randomness into the tutoring session. If a **randsubst** predicate has been defined for an operator, the first fact of the list will be replaced by the second fact on the list after the postconditions of that operator have been applied. The third element in the sublist is the value that determines the probability of occurrence of the random substitution. Either of the two facts can be "none" to allow addition or deletion of facts in the state description. The student is aware of a random substitution because a message is printed on the screen informing the occurrence of this change. The fourth element in the sublist is an additional optional message that gets printed on the screen after a random change.

20

For the multi-user tutor, we have defined an additional format for the randsubst predicate:

randsubst(< operator > ,[. . .,[< fact to be deleted > , < opt. message >],. . .]).

This format is required in the UNREP system when a **wait** operator is used in the knowledge base. In this case, the fact in the list is deleted from the true fact list; no probability weight is given because randomness is not a factor anymore. Using this special format, the random change message does not appear on the screen. An optional message may appear acknowledging the student on his decision to wait. This format is equivalent to a normal randsubst with a "none" second fact and a probability weight of one.

This format can better be explained using an example. Figure 1 shows the definitions for the operator **wait until delivery ship shoots gun line**. This operator may be applied by the student acting as the receiving ship OOD whenever his ship is already alongside and the delivery ship has not shot the gun line yet (precondition definition). The fact **gun line is shot** *cannot* be added in reality to the state description when the receiving ship student applies the **wait** operator, as the recommended and addpostcondition definitions suggest. The delivery ship must apply the operator **shoot gun line** in order to assert this fact in the state description. The reason why the wait operator seems to be asserting a fact that can only be added in reality by the delivery ship student, not by the receiving ship student, is because the METUTOR11 program must be able to solve the UNREP procedure completely before the tutoring session starts, to find if a solution path exists, for otherwise it would issue an error message advising there is no solution to the problem. The fact **gun line is shot** is deleted from the state description before the receiving ship student even gets to see it there, and a message acknowledges his decision to wait. In summary, this special **randsubst** format allows a

21

```
recommended([ shot(gun_line)], wait(until,delivery,ship,shoots,gun,line)).
precondition(wait(until,delivery,ship,shoots,gun,line),
    [alongside(ships),not shot(gun_line)]).
deletepostcondition(wait(until,delivery,ship,shoots,gun,line),[] .).
addpostcondition(wait(until,delivery,ship,shoots,gun,line),[shot(gun _line)]).
randsubst(wait(until,delivery,ship,shoots,gun,line),
    [[shoot(gun_line),"Please wait for gun line to be shot"]]).
```

Figure 1.   Example of "randsubst" Predicate

student to wait for an action that can only be performed by the other student. The state

description remains unchanged after a wait operator has been applied.

1.   UNREP Definitions

Following is a list of operators that may be applied by a student during a typical

UNREP tutoring session.   Each operator is described by giving first the type of ship

from which an OOD may apply the operator (either the receiving ship, or delivery ship,

or both), and next, the facts that are asserted in the state description once the operator

is applied.   The order in which the operators are listed is not necessarily the same in

which they are applied, and the order may vary from session to session.

- **Steer to ordered course and speed.**   Delivery ship OOD may apply it; accomplishes the fact **delivery ship is steady on ordered course and speed.**

- **Rig replenishment side.**   Both OODs may apply it; achieves the fact **unrep side is rigged on delivery (receiving) ship.**

- **Wait until delivery ship flies ROMEO at dip.**   Receiving ship OOD may apply it; does not assert any facts.

- **Fly ROMEO at dip on rigged side.**   Both ships may apply it; asserts **ROMEO flag is at dip on delivery (receiving) ship.**

- **Wait until delivery ship flies ROMEO close up.**   Receiving ship OOD.

- **Fly ROMEO close up.**   Both OODs may apply it; achieves **ROMEO flag is close up on delivery (receiving) ship, delivery ship is ready to receive,** and **receiving ship is ready to approach.**

- **Wait for receiving ship approach.**   Delivery ship OOD may apply it.

- **Approach delivery ship.**   Receiving ship OOD; asserts **ships are alongside.**

22

- **Wait until delivery ship shoots gun line.**   Receiving ship.

- **Shoot gun line.**   Delivery ship OOD; accomplishes **gun line is shot.**

- **Wait until receiving ship secures the line.**   Delivery ship OOD.

- **Receive and secure gun line.**   Receiving ship OOD; achieves **first line is secured**

- **Haul down ROMEO.**   Both ships; asserts **ROMEO flag is hauled down on delivery (receiving) ship.**

- **Fly BRAVO at fore.**   Both ships; accomplishes **BRAVO flag is at fore on delivery (receiving) ship.** When the receiving ship OOD applies it, a random substitution may occur, asserting the fact **delivery ship is bearing ahead.**

- **Wait until receiving ship flies PREP at dip.**   Delivery ship OOD.

- **Fly PREP at dip.**   Receiving ship OOD; accomplishes **receiving ship is ready to disengage.** A random substitution may also assert **distance between ships is changing.**

- **Announce fifteen minutes to disengage.**   Delivery ship OOD; asserts **fifteen minutes to disengage is announced.** A random change may achieve **unrep ships are on emergency** due to a steering system failure.

- **Wait until receiving ship flies PREP close up.**   Delivery ship.

- **Fly PREP close up.**   Receiving ship OOD; accomplishes **receiving ship is starting to disengage now,** and **PREP flag is close up on receiving ship.**

- **Disengage.**   Both ships may apply this operator; by applying it, the goal state **disengage is complete** is achieved.

- **Breakaway.**   Both ships. By applying it, the fact **ships are ok** is achieved; very useful operator when the ships are on emergency. Also, the goal state is reached when this operator is applied.

- **Announce emergency.**   Both ships; accomplishes **emergency is announced on delivery (receiving) ship.**

- **Fly EMERGENCY close up.**   Both ships; achieves **EMERGENCY flag is close up on delivery (receiving) ship.**

- **Increase speed.**   The receiving ship OOD may apply this operator. It should be used when the **delivery ship is bearing ahead,** so that the fact **receiving ship is on station** can be achieved.

- **Decrease speed.**   This operator is meant to cause a tricky situation for the receiving ship OOD. When the **delivery ship is bearing ahead,** if the receiving ship OOD applies this operator, instead of getting back on station, he will accomplish the fact **unrep ships are on emergency.** This operator offers the student an alternate path, but it happens to be the wrong path, therefore the student will have to get in track again by fixing the emergency first.

- **Issue rudder command.**   Receiving ship OOD. This operator may be used to achieve the fact **distance between ships is ok** when the **distance between ships is changing.** A random substitution may also assert the fact **unrep ships are on emergency** due to a steering system failure.

23

C. METUTOR11: A MULTI-USER APPROACH

1. General Description

The implementation of the UNREP system is based on the assumption that two students are tutored on the Underway Replenishment procedures at the same time. One of the students is playing the role of an OOD aboard a delivery ship, while the other student acts as the OOD aboard a receiving ship.

The UNREP tutor system was tested using the PDSS (Program Development Support System, by Logicware Inc.) facility in the Optimum V Workstations. A typical tutoring session is started by having each student load two files in the PDSS system. The first file is the METUTOR11 and the second file is either the MEUNREPDEL, for the delivery-ship-OOD student, or the MEUNREPREC, in the case of the receiving-ship-OOD student. Then, each student queries the predicate **go.** By doing this, the top level of the means-ends tutorial module is activated in the METUTOR11, therefore invoking the system. The tutor then prints on the screen an introduction[5] and then the tutoring session starts. The tutor shows the student a list of all the true facts at a given state and prompts him to choose an operator. The student then types in his choice of operator and waits for the tutor's next request. Three things can happen at this point: first, the tutor may accept the student's operator and apply it to the state in the simulation; second, the tutor may refuse an erroneous operator and ask the student for another; third, it may send a message to the student informing him that the chosen operator was ignored by the tutor because the state was changed by the other student's actions, and to please select an operator again.

The third alternative is a characteristic of the multi-user version of the METUTOR11 program. The tutor has to analyze the situation again, based on the facts describing the new state. Basically, of the two students, whoever enters first his choice

---

5 See appendices A and B for sample test runs.

of operator gets to continue the tutoring session normally without interruptions, while the other has to enter a new choice of operator. While some facts in the state description can be asserted by any of the two students, others can only be asserted specifically by only one of the two students. Each student will reach different points in the tutoring session where they must wait until the other student applies an operator that will achieve the state description needed by the first student to continue his session.

2. Implementation

The UNREP system was implemented by modifying and adding some rules to the original METUTOR11 program. The main implementation characteristic of the UNREP system is a single file common to both students using the tutor. Stored in this file are a session's current time-stamp and state description.

The file used in the UNREP system is called **FACTS**. Figure 2 shows an example of the contents of this file for a given state. This file is initialized at the top level of the means-ends tutor with an empty list as the state description (since the fact predicates are defined so that there are not any true facts yet at this point) and a time-stamp equal to zero. Also, a time-stamp predicate is asserted in the tutor's database with an initial value of zero.

After the two halves of the tutor program have been activated on two terminals, the first student to input his choice of operator adds some facts to his current state description and the time-stamp is incremented by one. The updated time-stamp and the new state description are saved in the FACTS file; also the incremented time-stamp value is asserted in the tutor's database for the first student, replacing the old value. When the second student tries to input his choice of operator, his tutor program first compares the FACTS file time-stamp to the time-stamp fact asserted in the tutor's database for the second student. When both time-stamps agree, the tutor accepts the

```
15    /* this is the current value of the time-stamp */

[ok(unrep_ships),ok(distance_between_ships),rigged_on delivery_ship(unrep_side)]

      /* and this is the current state description */
```

Figure 2.   Sample FACTS File Contents

second student's choice of operator and continues tutoring, but, since the first student

changed the time-stamp:

1. a **time-stamp violation** predicate is asserted in the second tutor's database;

2. the second tutor ignores the operator chosen by the second student;

3. it reads the new state and the new time-stamp from the FACTS file;

4. it prints a message informing the student of the anomaly, and lists for him the new state description; and

5. it analyzes the problem again to determine the best operator now, based on the new state description, before prompting the student for a new operator.

The behavior just described is implemented using Prolog backtracking in the

METUTOR11 program. Figure 3 shows the actual implementation details that allow

multi-user tutoring through time-stamp coordination. This code is mixed with the

means-ends tutor code. When the recursive means-ends program pauses to ask the

student his next choice of operator, the **check_with_student** predicate queries the

check_time_stamp to compare the FACTS file time-stamp against the database's time-

stamp (after reading the student's input from the screen). If both time-stamps agree, the

check_with_student rule succeeds, causing the **met** rule to   continue with   its normal

sequence:

1. delete and add postconditions,

2. perform random substitutions,

26

```
met(STATE,GOAL,OPLIST,GOALSTATE,STACK,PRELIST,PREOPLIST,...) :-
   check_with_student(OP,PRESTATE,D,NEWOP),
     (additional code )
   time_stamp(TS),
   retract(time_stamp(TS)),
   NTS is TS+1,
   asserta(time_stamp(NTS)),
   update_fact_file(NTS,POSTLIST),!,
   means_ends_tutor(POSTLIST,GOAL,POSTOPLIST,GOALSTATE,...).
met(STATE,GOAL,OPLIST,GOALSTATE,STACK,PRELIST,PROPLIST,...) :-
   ts_violation,retract(ts_violation),
   retract(time_stamp(TS)),nl,
   write("*** SORRY,IGNORED YOUR OPERATOR. OTHER USER JUST $
SCHANGED THE TRUE FACT LIST"),nl,
   write("here comes the new state:"),nl,
   read_fact_file(NTS,NEWSTATE),
   asserta(time_stamp(NTS)),
   means_ends_tutor(NEWSTATE,GOAL,OPLIST,GOALSTATE,STACK,
   GOALSTACK) .
check_with_student(O,S,D,NO) :-
   write(****************************),nl,
   write("The following facts are now true:"),nl,
   writelist(S,state),write("."),nl,
   write("What operator do you choose?"),nl,
   niceread(AO2),!,check_time_stamp,
   space_parse(AO2,O3),!,handle_student_op(O3,O,S,D,NO).
check_time_stamp :-
   read_fact_file(TS,S),time_stamp(TS1),
   asserta(ts_violation),!,compare( = ,TS,TS1),
   retract(ts_violation).
update_fact_file(NTS,NS) :-
   set_channel(outfile(outf),name = "facts.pro"),
   set_channel(outfile(outf),buffer = 1000),
   set_output(outfile(outf)),
   write(NTS),nl,write(NS),
   close_output(outfile(outf)).
read_fact_file(TS,S) :-
   set_channel(infile(d),name = "work/salgado/meunrep.pro"),
   set_channel(infile(inf),name = "facts.pro"),
   set_input(infile(inf),skip_unread_input,
   read(TS),read(S),close_input(infile(inf)).
```

Figure 3.   Multi-user Implementation Details

3. save the new time-stamp (after incrementing it) in the FACTS file, along with the
   new state description (after the tutor has applied the student's operator), and,

4. recursively call **means_ends_tutor**.

27

If there exists a disagreement between the time-stamps, the **check_time_stamp** rule fails (not without asserting first the **time_stamp_violation** predicate in the tutor's database), and so do the **check_with_student** and the originating **met** rules, forcing the program to try the next **met** definition. In that case, the following steps take place:

1. **ts_violation** is retracted from the database;

2. a message is sent to the student informing him that his operator has been ignored due to a change of state;

3. the new time-stamp and state description are retrieved from the FACTS file;

4. the new time-stamp is asserted in the database, and,

5. the new state description is included in the recursive call to **means_ends_tutor**.

V. RESULTS

A. MEMORY REQUIREMENTS

The required memory space for the UNREP system was 42K bytes on each of the two computers. The breakdown for each component file was as follows:

METUTOR11	34000 bytes
MEUNREPDEL	7200 bytes
MEUNREPREC	7800 bytes
FACTS	100 bytes (maximum)

(Each student uses only one of the MEUNREPDEL and MEUNREPREC).

The amount of required memory space was insignificant compared to the space available in the ISI workstations. If we wanted to adapt the UNREP system to work on a personal computer, additional memory would be required for a Prolog interpreter or compiler. To give an idea, the Prolog-86 interpreter requires approximately 100K bytes of memory [Ref. 13: p. 1]; even then, the UNREP system plus this interpreter could easily fit in a floppy disk.

B. TIME CONSIDERATIONS

With an interactive program such as the UNREP system, the total CPU time required for a tutoring session is not a meaningful measure of performance, because the run-time can vary considerably from one student to another, depending on their knowledge of the subject matter and their learning ability; a student's typing skills can greatly affect the total run-time since that can increase the amount of time spent by the tutor checking for input errors. A tutoring session could last as little as five minutes, if both students always typed in coordinated and correct answers, or as long as 45 minutes if the

29

students did not have any idea of what an Underway Replenishment was all about, and they did not have typing skills.

A more meaningful measure of time performance is the time required by the tutor to reply to a student. The time intervals, also called answering times, were measured starting from the moment a student entered his choice of operator (by depressing the enter or return key), until the moment the tutor replied a message on the screen. The shortest answering time was of approximately 1 second (on the average), for the cases where the student entered the correct operator and the tutor accepted it with an OK, and also when the student entered a nonsensical string of characters and the tutor replied with the message "Sorry, ignored your operator....". The longest answering times occurred when the student entered an operator which caused the inference engine to deeply analyze the student's hypothetical state (the resulting state description if the student's operator was to be applied) with calls to means-ends; in this case, the answering time could be anywhere between 5 and 30 seconds, depending on the severity of the student's misconception. This extended answering time was also true when the operator entered by the student had some spelling errors, but this time was less than for the abo. ~ mentioned case. These answering time measures are important in ICAI applications since long periods of time waiting for a reply can decrease the student's interest in learning.

Another consideration is that of the time required by the tutor to read and write data into the common file FACTS. It is important to avoid situations where one student's tutor would take a current state description, call it state-1, and use it to come up with a recommended operator, while meanwhile the second student's tutor could apply an operator and change state-1 into state-2; the recommended operator for student 1 would no longer be applicable. Fortunately, the time required to read a sample FACTS file was approximately 0.22 seconds, and 0.1 seconds to write, meaning that the times were small compared to the times usually taken by the students to input operators.

Thorough testing was performed on the system to detect possible deadlocks or any other system malfunctions due to simultaneous (or at about the same time) inputs from both students. Although we did not witness any problems during several test runs, this does not prove that they will not occur, it just makes them unlikely.

C. ACCURACY

We were concerned with the accuracy of the tutor's responses to a student's input. We analyzed cases where a student would make spelling errors on the input operators, or input a nonsense string of characters, or apply operators that did not agree with the tutor's recommended operator. In all cases it was found that the system would tutor as expected. During the first development stages of this thesis research, it was found that some tutoring statements made by the tutor did not make any sense, but it was only because the knowledge representation had some bugs, therefore causing a misinterpretation by the domain-reasoning methods.

D. PROBLEM COMPLEXITY

The application domain of the tutor is very complicated in reality. The factors involved in a real-situation Underway Replenishment can be numerous and varied in type. For purposes of this thesis, the knowledge representation was kept small to allow enough time to develop and test the system during nine months. Even then, the steps required by a student to complete a successful UNREP procedure were 13 for both the delivery and receiving ship students, assuming that no emergencies have occurred, the students have applied all the "wait" operators, and they have not made any mistakes.

VI. CONCLUSIONS

A. MAJOR ACHIEVEMENTS

This thesis has developed a prototype of a new kind of computer tutor for naval training. Traditional teaching methods involving classroom instruction can provide a student with the general knowledge in Underway Replenishment procedures, but they cannot cover all the necessary coordination skills to execute this difficult task, between two Officers of the Deck (OOD's) aboard each of the ships. Alternatively, hands-on training is too expensive and may be dangerous. The UNREP tutor teaches Underway Replenishment by using Artificial Intelligence (AI) techniques in an Intelligent Computer-Assisted Instruction program. The system is capable of simulating an Underway Replenishment situation and training two students simultaneously on separate computer workstations, so that coordination skills are emphasized. Other ICAI systems have been developed for military applications, but the novelty of the UNREP system is its ability to handle two students at the same time on a joint application.

The memory space requirements of the system have shown that the UNREP tutor is portable and can be adapted to smaller computer systems to allow training of personnel aboard ships, without expensive or hard to reach equipment. The tutor runs sufficiently fast on the prototype machine to make speed not an issue.

B. WEAKNESSES AND RECOMMENDATIONS

Although this research accomplished its major objectives, some minor weaknesses were found during the testing stage of the system.

The knowledge base representation of the UNREP procedure involved most of the important steps of the overall operation, and even an extra touch of reality was

32

introduced to the system by adding some emergency situations. However, a larger knowledge base would be needed if the system was to be more realistic.

The number of operators, and therefore the number of definitions in the knowledge base, have a direct impact on the time of execution, so if we wanted to have a more realistic simulation by expanding the knowledge base, we would need to figure out how to speed up the execution of the tutor program. An alternative would be to compile the tutor instead of using the PDSS facility interpreter. Quintus Prolog offers such a compiler which is notable for its very high execution speed and its similarity to most Prolog syntaxes (MProlog among them).

The names of operators defined for this application were very long strings of characters. The use of menus to display the operators and choose them is recommended for future enhancements of this system.

Tutoring would be more effective if a graphics interface was added to the system. The current version of the UNREP system does not include actual distances between ships, bearings, speeds of ships, and colors of signal flags, among other necessary factors in a typical UNREP situation. These factors could easily be included in a graphics interface, therefore allowing better simulation and in-depth training.

APPENDIX A.   UNREP DELIVERY SHIP DEMONSTRATION

MPROLOG (2.1.0) LOGIC - LAB
(c) 1985 Logicware Inc.
PDSS Program Development Support System

```
: read metutor11
: read meunrepdel
: go.
```

This is a test of UNREP procedures (DELIVERY SHIP)

Your objectives: disengage is complete.
(Type h after asterisk prompt for help)

Wait a moment while I analyze the problem thoroughly.
****
The following facts are now true:
unrep_ships are ok, receiving_ship is on_station, and distance_between_ships
 are ok.
What operator do you choose?
*steer to orderd course and speed
I assume you mean steer(to,ordered,course,and,speed).
OK.
****
The following facts are now true:
delivery_ship is steady_on_ordered_course_and_speed, unrep_ships are ok,
receiving_ship is on_station, and distance_between_ships are ok.
What operator do you choose?
*fly romeo at dip on rigged side
I will try it, but it is not recommended first when unrep_side must be
rigged_on_delivery_ship and romeo_flag must be at_dip_on_delivery_ship.
****
The following facts are now true:
romeo_flag is at_dip_on_delivery_ship, delivery_ship is
steady_on_ordered_course_and_speed, unrep_ships are ok, receiving_ship is
on_station, and distance_between_ships are ok.
What operator do you choose?
*rig replenishemnt side

*** SORRY,IGNORED YOUR OPERATOR. OTHER USER JUST CHANGED THE TRUE FACT LIST
here comes the new state:
****
The following facts are now true:
romeo_flag is at_dip_on_receiving_ship, receiving_ship is ready_to_approach,
unrep_side is rigged_on_receiving_ship, romeo_flag is at_dip_on_delivery_ship,
delivery_ship is steady_on_ordered_course_and_speed, unrep_ships are ok,
receiving_ship is on_station, and distance_between_ships are ok.
What operator do you choose?

34

*rig replenishment side
OK.
****
The following facts are now true:
unrep_side is rigged_on_delivery_ship, romeo_flag is at_dip_on_receiving_ship,
receiving_ship is ready_to_approach, unrep_side is rigged_on_receiving_ship,
romeo_flag is at_dip_on_delivery_ship, delivery_ship is
steady_on_ordered_course_and_speed, unrep_ships are ok, receiving_ship is
on_station, and distance_between_ships are ok.
What operator do you choose?
*wait for receiving ship approach
That operator requires that delivery_ship must be ready_to_receive.
****
The following facts are now true:
unrep_side is rigged_on_delivery_ship, romeo_flag is at_dip_on_receiving_ship,
receiving_ship is ready_to_approach, unrep_side is rigged_on_receiving_ship,
romeo_flag is at_dip_on_delivery_ship, delivery_ship is
steady_on_ordered_course_and_speed, unrep_ships are ok, receiving_ship is
on_station, and distance_between_ships are ok.
What operator do you choose?
*fly romeo closed up
OK.
****
The following facts are now true:
delivery_ship is ready_to_receive, romeo_flag is closed_up_on_delivery_ship,
romeo_flag is at_dip_on_receiving_ship, receiving_ship is ready_to_approach,
unrep_side is rigged_on_receiving_ship, delivery_ship is
steady_on_ordered_course_and_speed, unrep_ships are ok, receiving_ship is
on_station, and distance_between_ships are ok.
What operator do you choose?
*wait for receiving ship approach
OK.

Please wait a moment until receiving ship is alongside
****
The following facts are now true:
delivery_ship is ready_to_receive, romeo_flag is closed_up_on_delivery_ship,
romeo_flag is at_dip_on_receiving_ship, receiving_ship is ready_to_approach,
unrep_side is rigged_on_receiving_ship, delivery_ship is
steady_on_ordered_course_and_speed, unrep_ships are ok, receiving_ship is
on_station, and distance_between_ships are ok.
What operator do you choose?
*wait for receiving ship approach

*** SORRY,IGNORED YOUR OPERATOR. OTHER USER JUST CHANGED THE TRUE FACT LIST
here comes the new state:
****
The following facts are now true:
ships are alongside, romeo_flag is closed_up_on_receiving_ship, delivery_ship
 is ready_to_receive, romeo_flag is closed_up_on_delivery_ship, unrep_side is
rigged_on_receiving_ship, delivery_ship is steady_on_ordered_course_and_speed,
unrep_ships are ok, receiving_ship is on_station, and distance_between_ships
 are ok.
What operator do you choose?
*shoot gun line
OK.

35

```
**********************************
The following facts are now true:
gun_line is shot, ships are alongside, romeo_flag is
closed_up_on_receiving_ship, delivery_ship is ready_to_receive, romeo_flag is
closed_up_on_delivery_ship, unrep_side is rigged_on_receiving_ship,
delivery_ship is steady_on_ordered_course_and_speed, unrep_ships are ok,
receiving_ship is on_station, and distance_between_ships are ok.
What operator do you choose?
*haul down romeo
That operator requires that first_line must be secured.
**********************************
The following facts are now true:
gun_line is shot, ships are alongside, romeo_flag is
closed_up_on_receiving_ship, delivery_ship is ready_to_receive, romeo_flag is
closed_up_on_delivery_ship, unrep_side is rigged_on_receiving_ship,
delivery_ship is steady_on_ordered_course_and_speed, unrep_ships are ok,
receiving_ship is on_station, and distance_between_ships are ok.
What operator do you choose?
*wait until receiving ship secures the line
OK.

Please wait a moment until receiving ship secures the first line
**********************************
The following facts are now true:
gun_line is shot, ships are alongside, romeo_flag is
closed_up_on_receiving_ship, delivery_ship is ready_to_receive, romeo_flag is
closed_up_on_delivery_ship, unrep_side is rigged_on_receiving_ship,
delivery_ship is steady_on_ordered_course_and_speed, unrep_ships are ok,
receiving_ship is on_station, and distance_between_ships are ok.
What operator do you choose?
*haul

*** SORRY,IGNORED YOUR OPERATOR. OTHER USER JUST CHANGED THE TRUE FACT LIST
here comes the new state:
**********************************
The following facts are now true:
romeo_flag is hauled_down_on_receiving_ship, first_line is secured, gun_line
 is shot, ships are alongside, delivery_ship is ready_to_receive, romeo_flag
 is closed_up_on_delivery_ship, unrep_side is rigged_on_receiving_ship,
delivery_ship is steady_on_ordered_course_and_speed, unrep_ships are ok,
receiving_ship is on_station, and distance_between_ships are ok.
What operator do you choose?
*haul down romeo
OK.
**********************************
The following facts are now true:
romeo_flag is hauled_down_on_delivery_ship, romeo_flag is
hauled_down_on_receiving_ship, first_line is secured, ships are alongside,
delivery_ship is ready_to_receive, unrep_side is rigged_on_receiving_ship,
delivery_ship is steady_on_ordered_course_and_speed, unrep_ships are ok,
receiving_ship is on_station, and distance_between_ships are ok.
What operator do you choose?
*fly bravo at fore
OK.
**********************************
The following facts are now true:
```

bravo_flag is at_fore_on_delivery_ship, romeo_flag is
hauled_down_on_delivery_ship, romeo_flag is hauled_down_on_receiving_ship,
first_line is secured, ships are alongside, delivery_ship is ready_to_receive,
unrep_side is rigged_on_receiving_ship, delivery_ship is
steady_on_ordered_course_and_speed, unrep_ships are ok, receiving_ship is
on_station, and distance_between_ships are ok.
What operator do you choose?
*wait until receiving ship flies prep at dip

*** SORRY,IGNORED YOUR OPERATOR. OTHER USER JUST CHANGED THE TRUE FACT LIST
here comes the new state:
****
The following facts are now true:
distance_between_ships are changing, receiving_ship is ready_to_disengage,
receiving_ship is on_station, bravo_flag is at_fore_on_receiving_ship,
bravo_flag is at_fore_on_delivery_ship, romeo_flag is
hauled_down_on_delivery_ship, romeo_flag is hauled_down_on_receiving_ship,
first_line is secured, ships are alongside, delivery_ship is ready_to_receive,
unrep_side is rigged_on_receiving_ship, delivery_ship is
steady_on_ordered_course_and_speed, and unrep_ships are ok.
What operator do you choose?
*announce fifteen min to disengage
OK.
****
The following facts are now true:
fifteen_min_to_disengage is announced, distance_between_ships are changing,
receiving_ship is ready_to_disengage, receiving_ship is on_station, bravo_flag
 is at_fore_on_receiving_ship, bravo_flag is at_fore_on_delivery_ship,
romeo_flag is hauled_down_on_delivery_ship, romeo_flag is
hauled_down_on_receiving_ship, first_line is secured, ships are alongside,
delivery_ship is ready_to_receive, unrep_side is rigged_on_receiving_ship,
delivery_ship is steady_on_ordered_course_and_speed, and unrep_ships are ok.
What operator do you choose?
*wait until receiving ship flies prep closed up
OK.

Please wait until the receiving ship starts to disengage
****
The following facts are now true:
fifteen_min_to_disengage is announced, distance_between_ships are changing,
receiving_ship is ready_to_disengage, receiving_ship is on_station, bravo_flag
 is at_fore_on_receiving_ship, bravo_flag is at_fore_on_delivery_ship,
romeo_flag is hauled_down_on_delivery_ship, romeo_flag is
hauled_down_on_receiving_ship, first_line is secured, ships are alongside,
delivery_ship is ready_to_receive, unrep_side is rigged_on_receiving_ship,
delivery_ship is steady_on_ordered_course_and_speed, and unrep_ships are ok.
What operator do you choose?
*wait until receiving ship

*** SORRY,IGNORED YOUR OPERATOR. OTHER USER JUST CHANGED THE TRUE FACT LIST
here comes the new state:
****
The following facts are now true:
receiving_ship is starting_to_disengage_now, prep_flag is
closed_up_on_receiving_ship, distance_between_ships are ok,
fifteen_min_to_disengage is announced, receiving_ship is ready_to_disengage,

37

receiving_ship is on_station, bravo_flag is at_fore_on_receiving_ship,
bravo_flag is at_fore_on_delivery_ship, romeo_flag is
hauled_down_on_delivery_ship, romeo_flag is hauled_down_on_receiving_ship,
first_line is secured, ships are alongside, delivery_ship is ready_to_receive,
unrep_side is rigged_on_receiving_ship, delivery_ship is
steady_on_ordered_course_and_speed, and unrep_ships are ok.
What operator do you choose?
*disengage
OK.

Yes
: bye
Normal exit from MPROLOG PDSS

APPENDIX B. UNREP RECEIVING SHIP DEMONSTRATION

```
                    MPROLOG (2.1.0) LOGIC - LAB
                        (c) 1985 Logicware Inc.
                PDSS Program Development Support System

: read metutor11
: read meunreprec
: go.
```

                   This is a test of UNREP procedures (RECEIVING SHIP)

Your objectives: disengage is complete.
(Type h after asterisk prompt for help)

Wait a moment while I analyze the problem thoroughly.
****
The following facts are now true:
unrep_ships are ok, receiving_ship is on_station, and distance_between_ships
 are ok.
What operator do you choose?
*rig replenishment side

*** SORRY,IGNORED YOUR OPERATOR. OTHER USER JUST CHANGED THE TRUE FACT LIST
here comes the new state:
****
The following facts are now true:
romeo_flag is at_dip_on_delivery_ship, delivery_ship is
steady_on_ordered_course_and_speed, unrep_ships are ok, receiving_ship is
on_station, and distance_between_ships are ok.
What operator do you choose?
*rig replenishment side
OK.
****
The following facts are now true:
unrep_side is rigged_on_receiving_ship, romeo_flag is at_dip_on_delivery_ship,
delivery_ship is steady_on_ordered_course_and_speed, unrep_ships are ok,
receiving_ship is on_station, and distance_between_ships are ok.
What operator do you choose?
*fly romeo at dip on rigged side
OK.
****
The following facts are now true:
romeo_flag is at_dip_on_receiving_ship, receiving_ship is ready_to_approach,
unrep_side is rigged_on_receiving_ship, romeo_flag is at_dip_on_delivery_ship,
delivery_ship is steady_on_ordered_course_and_speed, unrep_ships are ok,
receiving_ship is on_station, and distance_between_ships are ok.
What operator do you choose?
*wait until delivery ship flies romeo closed up

39

OK.
Please wait until the delivery ship is ready to receive
****
The following facts are now true:
romeo_flag is at_dip_on_receiving_ship, receiving_ship is ready_to_approach,
unrep_side is rigged_on_receiving_ship, romeo_flag is at_dip_on_delivery_ship,
delivery_ship is steady_on_ordered_course_and_speed, unrep_ships are ok,
receiving_ship is on_station, and distance_between_ships are ok.
What operator do you choose?
*fly romeo closed up

*** SORRY,IGNORED YOUR OPERATOR. OTHER USER JUST CHANGED THE TRUE FACT LIST
here comes the new state:
****
The following facts are now true:
delivery_ship is ready_to_receive, romeo_flag is closed_up_on_delivery_ship,
romeo_flag is at_dip_on_receiving_ship, receiving_ship is ready_to_approach,
unrep_side is rigged_on_receiving_ship, delivery_ship is
steady_on_ordered_course_and_speed, unrep_ships are ok, receiving_ship is
on_station, and distance_between_ships are ok.
What operator do you choose?
*fly romeo closed up
OK.
****
The following facts are now true:
romeo_flag is closed_up_on_receiving_ship, receiving_ship is
commencing_approach, delivery_ship is ready_to_receive, romeo_flag is
closed_up_on_delivery_ship, unrep_side is rigged_on_receiving_ship,
delivery_ship is steady_on_ordered_course_and_speed, unrep_ships are ok,
receiving_ship is on_station, and distance_between_ships are ok.
What operator do you choose?
*approach delivery ship
OK.
****
The following facts are now true:
ships are alongside, romeo_flag is closed_up_on_receiving_ship, delivery_ship
 is ready_to_receive, romeo_flag is closed_up_on_delivery_ship, unrep_side is
rigged_on_receiving_ship, delivery_ship is steady_on_ordered_course_and_speed,
unrep_ships are ok, receiving_ship is on_station, and distance_between_ships
 are ok.
What operator do you choose?
*wait until delivery ship shoots gun line
OK.

Please wait for gun line to be shot
****
The following facts are now true:
ships are alongside, romeo_flag is closed_up_on_receiving_ship, delivery_ship
 is ready_to_receive, romeo_flag is closed_up_on_delivery_ship, unrep_side is
rigged_on_receiving_ship, delivery_ship is steady_on_ordered_course_and_speed,
unrep_ships are ok, receiving_ship is on_station, and distance_between_ships
 are ok.
What operator do you choose?
*receive gun line

here comes the new state:
****
The following facts are now true:
gun_line is shot, ships are alongside, romeo_flag is
closed_up_on_receiving_ship, delivery_ship is ready_to_receive, romeo_flag is
closed_up_on_delivery_ship, unrep_side is rigged_on_receiving_ship,
delivery_ship is steady_on_ordered_course_and_speed, unrep_ships are ok,
receiving_ship is on_station, and distance_between_ships are ok.
What operator do you choose?
*receive gun line
Not a valid operator--please choose one of: approach delivery ship, disengage,
haul down romeo, decrease speed, rig replenishment side, wait(until,delivery,
ship,flies,romeo,at,dip), increase speed, announce emergency, receive(and,
secure,gun,line), wait(until,delivery,ship,flies,romeo,closed,up), fly(
emergency,closed,up), wait(until,delivery,ship,shoots,gun,line), fly(bravo,at,
fore), fly(romeo,closed,up), fly(romeo,at,dip,on,rigged,side), fly(prep,at,dip)
, issue rudder command, breakaway, and fly(prep,closed,up).
****
The following facts are now true:
gun_line is shot, ships are alongside, romeo_flag is
closed_up_on_receiving_ship, delivery_ship is ready_to_receive, romeo_flag is
closed_up_on_delivery_ship, unrep_side is rigged_on_receiving_ship,
delivery_ship is steady_on_ordered_course_and_speed, unrep_ships are ok,
receiving_ship is on_station, and distance_between_ships are ok.
What operator do you choose?
*receive and secure gun line
OK.
****
The following facts are now true:
first_line is secured, gun_line is shot, ships are alongside, romeo_flag is
closed_up_on_receiving_ship, delivery_ship is ready_to_receive, romeo_flag is
closed_up_on_delivery_ship, unrep_side is rigged_on_receiving_ship,
delivery_ship is steady_on_ordered_course_and_speed, unrep_ships are ok,
receiving_ship is on_station, and distance_between_ships are ok.
What operator do you choose?
*fly bravo at fore
That operator requires that romeo_flag must be hauled_down_on_receiving_ship.
****
The following facts are now true:
first_line is secured, gun_line is shot, ships are alongside, romeo_flag is
closed_up_on_receiving_ship, delivery_ship is ready_to_receive, romeo_flag is
closed_up_on_delivery_ship, unrep_side is rigged_on_receiving_ship,
delivery_ship is steady_on_ordered_course_and_speed, unrep_ships are ok,
receiving_ship is on_station, and distance_between_ships are ok.
What operator do you choose?
*haul down romeo
OK.
****
The following facts are now true:
romeo_flag is hauled_down_on_receiving_ship, first_line is secured, gun_line
 is shot, ships are alongside, delivery_ship is ready_to_receive, romeo_flag
 is closed_up_on_delivery_ship, unrep_side is rigged_on_receiving_ship,
delivery_ship is steady_on_ordered_course_and_speed, unrep_ships are ok,
receiving_ship is on_station, and distance_between_ships are ok.
What operator do you choose?

41

*fly bravo at fore
*** SORRY,IGNORED YOUR OPERATOR. OTHER USER JUST CHANGED THE TRUE FACT LIST
here comes the new state:
****
The following facts are now true:
bravo_flag is at_fore_on_delivery_ship, romeo_flag is
hauled_down_on_delivery_ship, romeo_flag is hauled_down_on_receiving_ship,
first_line is secured, ships are alongside, delivery_ship is ready_to_receive,
unrep_side is rigged_on_receiving_ship, delivery_ship is
steady_on_ordered_course_and_speed, unrep_ships are ok, receiving_ship is
on_station, and distance_between_ships are ok.
What operator do you choose?
*fly bravo at fore
OK.
Random change made: fact delivery_ship is bearing_ahead added, and
fact receiving_ship is on_station removed.
The delivery ship is bearing slightly ahead of your ship. What do you do?
****
The following facts are now true:
delivery_ship is bearing_ahead, bravo_flag is at_fore_on_receiving_ship,
bravo_flag is at_fore_on_delivery_ship, romeo_flag is
hauled_down_on_delivery_ship, romeo_flag is hauled_down_on_receiving_ship,
first_line is secured, ships are alongside, delivery_ship is ready_to_receive,
unrep_side is rigged_on_receiving_ship, delivery_ship is
steady_on_ordered_course_and_speed, unrep_ships are ok, and
distance_between_ships are ok.
What operator do you choose?
*increase speed
OK.
****
The following facts are now true:
receiving_ship is on_station, bravo_flag is at_fore_on_receiving_ship,
bravo_flag is at_fore_on_delivery_ship, romeo_flag is
hauled_down_on_delivery_ship, romeo_flag is hauled_down_on_receiving_ship,
first_line is secured, ships are alongside, delivery_ship is ready_to_receive,
unrep_side is rigged_on_receiving_ship, delivery_ship is
steady_on_ordered_course_and_speed, unrep_ships are ok, and
distance_between_ships are ok.
What operator do you choose?
*fly prep at dip
OK.
Random change made: fact distance_between_ships are changing added, and
fact distance_between_ships are ok removed.
The distance line shows a slight change. What should you do?
****
The following facts are now true:
distance_between_ships are changing, receiving_ship is ready_to_disengage,
receiving_ship is on_station, bravo_flag is at_fore_on_receiving_ship,
bravo_flag is at_fore_on_delivery_ship, romeo_flag is
hauled_down_on_delivery_ship, romeo_flag is hauled_down_on_receiving_ship,
first_line is secured, ships are alongside, delivery_ship is ready_to_receive,
unrep_side is rigged_on_receiving_ship, delivery_ship is
steady_on_ordered_course_and_speed, and unrep_ships are ok.
What operator do you choose?
*issue rudder command

*** SORRY,IGNORED YOUR OPERATOR. OTHER USER JUST CHANGED THE TRUE FACT LIST
here comes the new state:
****
The following facts are now true:
fifteen_min_to_disengage is announced, distance_between_ships are changing,
receiving_ship is ready_to_disengage, receiving_ship is on_station, bravo_flag
 is at_fore_on_receiving_ship, bravo_flag is at_fore_on_delivery_ship,
romeo_flag is hauled_down_on_delivery_ship, romeo_flag is
hauled_down_on_receiving_ship, first_line is secured, ships are alongside,
delivery_ship is ready_to_receive, unrep_side is rigged_on_receiving_ship,
delivery_ship is steady_on_ordered_course_and_speed, and unrep_ships are ok.
What operator do you choose?
*issue rudder command
OK.
****
The following facts are now true:
distance_between_ships are ok, fifteen_min_to_disengage is announced,
receiving_ship is ready_to_disengage, receiving_ship is on_station, bravo_flag
 is at_fore_on_receiving_ship, bravo_flag is at_fore_on_delivery_ship,
romeo_flag is hauled_down_on_delivery_ship, romeo_flag is
hauled_down_on_receiving_ship, first_line is secured, ships are alongside,
delivery_ship is ready_to_receive, unrep_side is rigged_on_receiving_ship,
delivery_ship is steady_on_ordered_course_and_speed, and unrep_ships are ok.
What operator do you choose?
*fly prep closed up
OK.
****
The following facts are now true:
receiving_ship is starting_to_disengage_now, prep_flag is
closed_up_on_receiving_ship, distance_between_ships are ok,
fifteen_min_to_disengage is announced, receiving_ship is ready_to_disengage,
receiving_ship is on_station, bravo_flag is at_fore_on_receiving_ship,
bravo_flag is at_fore_on_delivery_ship, romeo_flag is
hauled_down_on_delivery_ship, romeo_flag is hauled_down_on_receiving_ship,
first_line is secured, ships are alongside, delivery_ship is ready_to_receive,
unrep_side is rigged_on_receiving_ship, delivery_ship is
steady_on_ordered_course_and_speed, and unrep_ships are ok.
What operator do you choose?
*disengage

*** SORRY,IGNORED YOUR OPERATOR. OTHER USER JUST CHANGED THE TRUE FACT LIST
here comes the new state:

Yes
: bye
Normal exit from MPROLOG PDSS

APPENDIX C.   UNREP DELIVERY SHIP KNOWLEDGE BASE

```
module meunrepdel.

/*$eject*/
body.

intro('        This is a test of UNREP procedures (DELIVERY SHIP)') .

go :-
      tutor([ok(unrep_ships),
             on_station(receiving_ship),ok(distance_between_ships)],
             [complete(disengage)]) .


recommended([ok(unrep_ships)],breakaway) .
precondition(breakaway,[on_emergency(unrep_ships),
             announced_on_delivery_ship(emergency),
             closed_up_on_delivery_ship(emergency_flag)]) .
deletepostcondition(breakaway,[on_emergency(unrep_ships),
                     announced_on_delivery_ship(emergency),
                     closed_up_on_delivery_ship(emergency_flag)]) .
addpostcondition(breakaway,[ok(unrep_ships),complete(disengage)]) .


recommended([announced_on_delivery_ship(emergency)],announce(emergency)) .
precondition(announce(emergency),[on_emergency(unrep_ships)]) .
deletepostcondition(announce(emergency),[]) .
addpostcondition(announce(emergency),[announced_on_delivery_ship(emergency)]) .


recommended([closed_up_on_delivery_ship(emergency_flag)],
            fly(emergency,closed,up)) .
precondition(fly(emergency,closed,up),[on_emergency(unrep_ships)]) .
deletepostcondition(fly(emergency,closed,up),
                    [at_fore_on_delivery_ship(bravo_flag)]) .
addpostcondition(fly(emergency,closed,up),
                 [closed_up_on_delivery_ship(emergency_flag)]) .




recommended([steady_on_ordered_course_and_speed(delivery_ship)],
            steer(to,ordered,course,and,speed)) .
precondition(steer(to,ordered,course,and,speed),[]) .
deletepostcondition(steer(to,ordered,course,and,speed),[]) .
addpostcondition(steer(to,ordered,course,and,speed),
                 [steady_on_ordered_course_and_speed(delivery_ship)]) .
```

44

```
recommended([rigged_on_delivery_ship(unrep_side)],rig(replenishment,side)) .
precondition(rig(replenishment,side),
             [not rigged_on_delivery_ship(unrep_side)]) .
deletepostcondition(rig(replenishment,side),[]) .
addpostcondition(rig(replenishment,side),
                 [rigged_on_delivery_ship(unrep_side)]) .


recommended([at_dip_on_delivery_ship(romeo_flag)],
            fly(romeo,at,dip,on,rigged,side)) .
precondition(fly(romeo,at,dip,on,rigged,side),
             [steady_on_ordered_course_and_speed(delivery_ship)]) .
deletepostcondition(fly(romeo,at,dip,on,rigged,side),[]) .
addpostcondition(fly(romeo,at,dip,on,rigged,side),
                 [at_dip_on_delivery_ship(romeo_flag)]) .


recommended([ready_to_receive(delivery_ship)],fly(romeo,closed,up)) .
precondition(fly(romeo,closed,up),[at_dip_on_delivery_ship(romeo_flag),
             rigged_on_delivery_ship(unrep_side)]) .
deletepostcondition(fly(romeo,closed,up),
                    [at_dip_on_delivery_ship(romeo_flag),
                     rigged_on_delivery_ship(unrep_side)]) .
addpostcondition(fly(romeo,closed,up),[ready_to_receive(delivery_ship),
                 closed_up_on_delivery_ship(romeo_flag)]) .


recommended([alongside(ships)],wait(for,receiving,ship,approach)) .
precondition(wait(for,receiving,ship,approach),
             [ready_to_receive(delivery_ship),not alongside(ships)]) .
deletepostcondition(wait(for,receiving,ship,approach),[]) .
addpostcondition(wait(for,receiving,ship,approach),
                 [alongside(ships)]) .
randsubst(wait(for,receiving,ship,approach),[[alongside(ships),
         "Please wait a moment until receiving ship is alongside"]]) .


recommended([shot(gun_line)],shoot(gun,line)) .
precondition(shoot(gun,line),
             [alongside(ships),ready_to_receive(delivery_ship)]) .
deletepostcondition(shoot(gun,line),[]) .
addpostcondition(shoot(gun,line),[shot(gun_line)]) .


recommended([secured(first_line)],wait(until,receiving,ship,secures,the,line)) .
precondition(wait(until,receiving,ship,secures,the,line),[shot(gun_line),
             not secured(first_line)]) .
deletepostcondition(wait(until,receiving,ship,secures,the,line),[]) .
addpostcondition(wait(until,receiving,ship,secures,the,line),
                 [secured(first_line)]) .
randsubst(wait(until,receiving,ship,secures,the,line),[[secured(first_line),
         "Please wait a moment until receiving ship secures the first line"]]) .


recommended([hauled_down_on_delivery_ship(romeo_flag)],haul(down,romeo)) .
precondition(haul(down,romeo),[secured(first_line),shot(gun_line),
```

```
                    ok(unrep_ships)]) .
deletepostcondition(haul(down,romeo),
                    [closed_up_on_delivery_ship(romeo_flag),shot(gun_line)]) .
addpostcondition(haul(down,romeo),[hauled_down_on_delivery_ship(romeo_flag)]) .


recommended([at_fore_on_delivery_ship(bravo_flag)],fly(bravo,at,fore)) .
precondition(fly(bravo,at,fore),[hauled_down_on_delivery_ship(romeo_flag),
            ok(unrep_ships)]) .
deletepostcondition(fly(bravo,at,fore),[]) .
addpostcondition(fly(bravo,at,fore),[at_fore_on_delivery_ship(bravo_flag)]) .


recommended([ready_to_disengage(receiving_ship)],
            wait(until,receiving,ship,flies,prep,at,dip)) .
precondition(wait(until,receiving,ship,flies,prep,at,dip),
            [at_fore_on_delivery_ship(bravo_flag),ok(unrep_ships),
            not at_dip_on_receiving_ship(prep_flag)]) .
deletepostcondition(wait(until,receiving,ship,flies,prep,at,dip),[]) .
addpostcondition(wait(until,receiving,ship,flies,prep,at,dip),
                [ready_to_disengage(receiving_ship)]) .
randsubst(wait(until,receiving,ship,flies,prep,at,dip),
        [[ready_to_disengage(receiving_ship),
        "Please wait until the receiving ship flies prep at dip"]]) .


recommended([announced(fifteen_min_to_disengage)],
            announce(fifteen,min,to,disengage)) .
precondition(announce(fifteen,min,to,disengage),
            [at_fore_on_delivery_ship(bravo_flag),ok(unrep_ships),
            ready_to_disengage(receiving_ship)]) .
deletepostcondition(announce(fifteen,min,to,disengage),[]) .
addpostcondition(announce(fifteen,min,to,disengage),
                [announced(fifteen_min_to_disengage)]) .
randsubst(announce(fifteen,min,to,disengage),[[ok(unrep_ships),
        on_emergency(unrep_ships),5.0e-1,"Steering system failure. $
        $ What should you do now?"]]) .


recommended([starting_to_disengage_now(receiving_ship)],
            wait(until,receiving,ship,flies,prep,closed,up)) .
precondition(wait(until,receiving,ship,flies,prep,closed,up),
            [announced(fifteen_min_to_disengage),ok(unrep_ships),
            not closed_up_on_receiving_ship(prep_flag)]) .
deletepostcondition(wait(until,receiving,ship,flies,prep,closed,up),[]) .
addpostcondition(wait(until,receiving,ship,flies,prep,closed,up),
                [starting_to_disengage_now(receiving_ship)]) .
randsubst(wait(until,receiving,ship,flies,prep,closed,up),
        [[starting_to_disengage_now(receiving_ship),
        "Please wait until the receiving ship starts to disengage"]]) .


recommended([complete(disengage)],disengage) .
precondition(disengage,[starting_to_disengage_now(receiving_ship),
            ok(unrep_ships),announced(fifteen_min_to_disengage)]) .
deletepostcondition(disengage,[announced(fifteen_min_to_disengage),
```

```
                    at_fore_on_delivery_ship(bravo_flag),
          secured(first_line),alongside(ships)]) .
addpostcondition(disengage,[complete(disengage)]) .


nopref(steer(to,ordered,course,and,speed),rig(replenishment,side)) .
nopref(announce(emergency),fly(emergency,closed,up)) .

deletepostcondition_args(2) .

addpostcondition_args(2) .

endmod /* meunrepdel */   .
```

APPENDIX D.   UNREP RECEIVING SHIP KNOWLEDGE BASE


```
module meunreprec.

/*$eject*/
body.

intro('          This is a test of UNREP procedures (RECEIVING SHIP)').

go :-
        tutor([ok(unrep_ships),
                on_station(receiving_ship),ok(distance_between_ships)],
                [complete(disengage)]) .


recommended([ok(unrep_ships)],breakaway).
precondition(breakaway,[on_emergency(unrep_ships),
                announced_on_receiving_ship(emergency),
              closed_up_on_receiving_ship(emergency_flag)]).
deletepostcondition(breakaway,[on_emergency(unrep_ships),
                    announced_on_receiving_ship(emergency),
                    alongside(ships),
                    closed_up_on_receiving_ship(emergency_flag)]).
addpostcondition(breakaway,[ok(unrep_ships),complete(disengage)]).


recommended([announced_on_receiving_ship(emergency)],announce(emergency)).
precondition(announce(emergency),[on_emergency(unrep_ships)]).
deletepostcondition(announce(emergency),[]).
addpostcondition(announce(emergency),[announced_on_receiving_ship(emergency)]).


recommended([closed_up_on_receiving_ship(emergency_flag)],
            fly(emergency,closed,up)).
precondition(fly(emergency,closed,up),[on_emergency(unrep_ships)]).
deletepostcondition(fly(emergency,closed,up),[]).
addpostcondition(fly(emergency,closed,up),
                    [closed_up_on_receiving_ship(emergency_flag)]).


recommended([on_station(receiving_ship)],increase(speed)).
precondition(increase(speed),[bearing_ahead(delivery_ship),ok(unrep_ships)]).
deletepostcondition(increase(speed),[bearing_ahead(delivery_ship)]).
addpostcondition(increase(speed),[on_station(receiving_ship)]).


recommended([on_station(receiving_ship)],decrease(speed)).
precondition(decrease(speed),[bearing_ahead(delivery_ship)]).
deletepostcondition(decrease(speed),[],[ok(unrep_ships)],
   "The operator you just applied made$
```

48

```
$ the whole situation worst than before. Now there is an emergency!").
addpostcondition(decrease(speed),[on_emergency(unrep_ships)]).


recommended([ok(distance_between_ships)],issue(rudder,command)).
precondition(issue(rudder,command),[changing(distance_between_ships)]).
deletepostcondition(issue(rudder,command),[changing(distance_between_ships)]).
addpostcondition(issue(rudder,command),[ok(distance_between_ships)]).
randsubst(issue(rudder,command),[[ok(unrep_ships),
    on_emergency(unrep_ships),4.0e-1,"Steering system failure.   What$
    $ should you do now?!"]]).




recommended([rigged_on_receiving_ship(unrep_side)],rig(replenishment,side)).
precondition(rig(replenishment,side),
            [not rigged_on_receiving_ship(unrep_side)]).
deletepostcondition(rig(replenishment,side),[]).
addpostcondition(rig(replenishment,side),
                [rigged_on_receiving_ship(unrep_side)]).


recommended([at_dip_on_delivery_ship(romeo_flag)],
            wait(until,delivery,ship,flies,romeo,at,dip)).
precondition(wait(until,delivery,ship,flies,romeo,at,dip),
            [not alongside(ships),
            not at_dip_on_delivery_ship(romeo_flag)]).
deletepostcondition(wait(until,delivery,ship,flies,romeo,at,dip),[]).
addpostcondition(wait(until,delivery,ship,flies,romeo,at,dip),
                [at_dip_on_delivery_ship(romeo_flag)]).
randsubst(wait(until,delivery,ship,flies,romeo,at,dip),
        [[at_dip_on_delivery_ship(romeo_flag),
        "Please wait a moment until delivery ship flies romeo at the dip"]]).

recommended([ready_to_approach(receiving_ship)],
            fly(romeo,at,dip,on,rigged,side)).
precondition(fly(romeo,at,dip,on,rigged,side),
            [rigged_on_receiving_ship(unrep_side),
            at_dip_on_delivery_ship(romeo_flag)]).
deletepostcondition(fly(romeo,at,dip,on,rigged,side),[]).
addpostcondition(fly(romeo,at,dip,on,rigged,side),
                [at_dip_on_receiving_ship(romeo_flag),
                ready_to_approach(receiving_ship)]).


recommended([closed_up_on_delivery_ship(romeo_flag)],
            wait(until,delivery,ship,flies,romeo,closed,up)).
precondition(wait(until,delivery,ship,flies,romeo,closed,up),
            [ready_to_approach(receiving_ship),
            not closed_up_on_delivery_ship(romeo_flag)]).
deletepostcondition(wait(until,delivery,ship,flies,romeo,closed,up),[]).
addpostcondition(wait(until,delivery,ship,flies,romeo,closed,up),
                [closed_up_on_delivery_ship(romeo_flag)]).
randsubst(wait(until,delivery,ship,flies,romeo,closed,up),
        [[closed_up_on_delivery_ship(romeo_flag),
        "Please wait until the delivery ship is ready to receive"]]).
```

49

```
recommended([commencing_approach(receiving_ship)],fly(romeo,closed,up)).
precondition(fly(romeo,closed,up),[ready_to_approach(receiving_ship),
          closed_up_on_delivery_ship(romeo_flag)]).
deletepostcondition(fly(romeo,closed,up),[at_dip_on_receiving_ship(romeo_flag),
                ready_to_approach(receiving_ship)]).
addpostcondition(fly(romeo,closed,up),[closed_up_on_receiving_ship(romeo_flag),
            commencing_approach(receiving_ship)]).


recommended([alongside(ships)],approach(delivery,ship)).
precondition(approach(delivery,ship),[commencing_approach(receiving_ship)]).
deletepostcondition(approach(delivery,ship),
                [commencing_approach(receiving_ship)]).
addpostcondition(approach(delivery,ship),[alongside(ships)]).


recommended([shot(gun_line)],wait(until,delivery,ship,shoots,gun,line)).
precondition(wait(until,delivery,ship,shoots,gun,line),
          [alongside(ships),not shot(gun_line)]).
deletepostcondition(wait(until,delivery,ship,shoots,gun,line),[]).
addpostcondition(wait(until,delivery,ship,shoots,gun,line),[shot(gun_line)]).
randsubst(wait(until,delivery,ship,shoots,gun,line),
      [[shot(gun_line),"Please wait for gun line to be shot"]]).


recommended([secured(first_line)],receive(and,secure,gun,line)).
precondition(receive(and,secure,gun,line),[alongside(ships),shot(gun_line)]).
deletepostcondition(receive(and,secure,gun,line),[]).
addpostcondition(receive(and,secure,gun,line),[secured(first_line)]).


recommended([hauled_down_on_receiving_ship(romeo_flag)],haul(down,romeo)).
precondition(haul(down,romeo),[secured(first_line)]).
deletepostcondition(haul(down,romeo),[closed_up_on_receiving_ship(romeo_flag)]).
addpostcondition(haul(down,romeo),[hauled_down_on_receiving_ship(romeo_flag)]).


recommended([at_fore_on_receiving_ship(bravo_flag)],fly(bravo,at,fore)).
precondition(fly(bravo,at,fore),[hauled_down_on_receiving_ship(romeo_flag)]).
deletepostcondition(fly(bravo,at,fore),[]).
addpostcondition(fly(bravo,at,fore),[at_fore_on_receiving_ship(bravo_flag)]).
randsubst(fly(bravo,at,fore),[[on_station(receiving_ship),
  bearing_ahead(delivery_ship),7.0e-1,"The delivery ship is bearing slightly $
  $ahead of your ship. What do you do?"]]).


recommended([ready_to_disengage(receiving_ship)],fly(prep,at,dip)).
precondition(fly(prep,at,dip),
          [at_fore_on_receiving_ship(bravo_flag),
           on_station(receiving_ship)]).
deletepostcondition(fly(prep,at,dip),[]).
addpostcondition(fly(prep,at,dip),[ready_to_disengage(receiving_ship)]).
randsubst(fly(prep,at,dip),[[ok(distance_between_ships),
  changing(distance_between_ships),6.5e-1,"The distance line shows a $
```

```
        changing(distance_between_ships),6.5e-1,"The distance line shows a $
        $slight change. What should you do?"]]).


recommended([starting_to_disengage_now(receiving_ship)],fly(prep,closed,up)).
precondition(fly(prep,closed,up),[ok(distance_between_ships),
            ready_to_disengage(receiving_ship)]).
deletepostcondition(fly(prep,closed,up),[]).
addpostcondition(fly(prep,closed,up),[starting_to_disengage_now(receiving_ship),
                closed_up_on_receiving_ship(prep_flag)]).


recommended([complete(disengage)],disengage).
precondition(disengage,[starting_to_disengage_now(receiving_ship)]).
deletepostcondition(disengage,[ready_to_disengage(receiving_ship),
                at_fore_on_receiving_ship(bravo_flag),
                secured(first_line),alongside(ships)]).
addpostcondition(disengage,[complete(disengage),hauled_down(prep_flag)]).


nopref(announce(emergency),fly(emergency,closed,up)).

deletepostcondition_args(2).
deletepostcondition_args(4).

addpostcondition_args(2).

endmod /* meunreprec */ .
```

APPENDIX E.   METUTOR11 SOURCE CODE

NOTE: The code contained herein was written by, and used with the permission
of Professor Neil C. Rowe of the Naval Postgraduate School.

* A preceding asterix means that the predicate has been modified
  by the author of this thesis.
+ A preceding plus sign means that the predicate has been added by
  the author of this thesis.

```
module metutor11.


export    (tutor / 2).

import    (addpostcondition / 2,addpostcondition / 3,
          addpostcondition / 4,addpostcondition_args / 1,
          deletepostcondition / 2,deletepostcondition / 4,
          deletepostcondition_args / 1,precondition / 2,randsubst / 2,
          recommended / 2).

/*$eject*/
body.

dynamic(readbuff/1).
dynamic(top_goal/1).
dynamic(op_list/1).
dynamic(top_solution/1).
dynamic(randseed/1).
dynamic(mainline_states/4).
dynamic(cached/4).
dynamic(debugflag/0).
dynamic(unsolvable/2).
+ dynamic(time_stamp/1).
+ dynamic(ts_violation/0).



/****  PROBLEM-INDEPENDENT CODE FOR MEANS-ENDS TUTORING  ****/

* tutor(STATE,GOAL) :-
      not check_obvious_errors, do_intro, write("Your objectives:  "),
        writelist(GOAL,state), write("."), nl,
        write("(Type h after asterisk prompt for help)"), nl(2),
        uniqueassert(top_goal(GOAL)), bagof(X,P^precondition(X,P),XL),
        uniqueassert(op_list(XL)),
        write("Wait a moment while I analyze the problem thoroughly."), nl,
        once_means_ends(STATE,GOAL,OPLIST2,GOALSTATE2),
        uniqueassert(top_solution(STATELIST)),
        del_all_statements(mainline_states/4),
```

```prolog
                update_fact_file(0,[]),
                asserta(time_stamp(0)),
                means_ends_tutor(STATE,GOAL,OPLIST,GOALSTATE,[],[]), nl,
                nl, ! .
tutor(STATE,GOAL) :-
        write("Too bad: a solution is now impossible."), nl, ! .


means_ends_tutor(STATE,GOAL,[],STATE,STACK,GOALSTACK) :-
        difference(GOAL,STATE,[]), ! .
means_ends_tutor(STATE,GOAL,OPLIST,STATE,STACK,GOALSTACK) :-
        member([STATE,GOAL],STACK), !, fail .
means_ends_tutor(STATE,GOAL,OPLIST,STATE,STACK,GOALSTACK) :-
        not once_means_ends(STATE,GOAL,OPLIST,GOALSTATE), !, fail .
means_ends_tutor(STATE,GOAL,OPLIST,GOALSTATE,STACK,GOALSTACK) :-
        difference(GOAL,STATE,D), applicable_op(D,OP),
            precondition(OP,PRELIST), all_achievable(STATE,PRELIST),
            apply_op(OP,STATE,STATE2),
            once_means_ends(STATE2,GOAL,OPLIST2,GOALSTATE2), !,
            means_ends_tutor(STATE,PRELIST,PREOPLIST,PRESTATE,[[STATE,GOAL] |
                STACK],[GOAL|GOALSTACK]), !,
            met(STATE,GOAL,OPLIST,GOALSTATE,STACK,PRELIST,PREOPLIST,PRESTATE,
                OP,D,GOALSTACK) .


met(STATE,GOAL,PREOPLIST,PRESTATE,STACK,PRELIST,PREOPLIST,PRESTATE,OP,D,
            GOALSTACK) :-
        difference(GOAL,PRESTATE,[]), ! .
met(STATE,GOAL,PREOPLIST,PRESTATE,STACK,PRELIST,PREOPLIST,PRESTATE,OP,D,
            GOALSTACK) :-
        higher_goal_achieved(GOALSTACK,PRESTATE), ! .
met(STATE,GOAL,OPLIST,GOALSTATE,STACK,PRELIST,PREOPLIST,PRESTATE,OP,D,
            GOALSTACK) :-
        difference(GOAL,PRESTATE,D2), not applicable_op(D2,OP), !,
            means_ends_tutor(PRESTATE,GOAL,OPLIST2,GOALSTATE,[],GOALSTACK),
            append(PREOPLIST,OPLIST2,OPLIST) .
* met(STATE,GOAL,OPLIST,GOALSTATE,STACK,PRELIST,PREOPLIST,PRESTATE,OP,D,
            GOALSTACK) :-
        writedebug8(OP),
            check_with_student(OP,PRESTATE,D,NEWOP),
            get_deletepostcondition(NEWOP,PRESTATE,DELETEPOSTLIST),
            print_optional_message_d(NEWOP,PRESTATE),
            deleteitems(DELETEPOSTLIST,PRESTATE,PRESTATE2),
            get_addpostcondition(NEWOP,PRESTATE,ADDPOSTLIST),
            print_optional_message_a(NEWOP,PRESTATE),
            union(ADDPOSTLIST,PRESTATE2,POSTLIST2),
            do_randsubst(NEWOP,POSTLIST2,POSTLIST),
            check_mainline_return(POSTLIST),time_stamp(TS),
            retract(time_stamp(TS)),NTS is TS+1,asserta(time_stamp(NTS)),
            update_fact_file(NTS,POSTLIST),!,
            means_ends_tutor(POSTLIST,GOAL,POSTOPLIST,GOALSTATE,[],[GOAL|
                GOALSTACK]), append(PREOPLIST,[NEWOP|POSTOPLIST],OPLIST) .
+ met(STATE,GOAL,OPLIST,GOALSTATE,STACK,PRELIST,PREOPLIST,PRESTATE,OP,D,
            GOALSTACK) :-
        ts_violation,retract(ts_violation),retract(time_stamp(TS)),
            nl,write("*** SORRY,IGNORED YOUR OPERATOR. OTHER $
            $USER JUST CHANGED THE TRUE FACT LIST"),nl,
```

```
            write("here comes the new state:"),nl,
            read_fact_file(NTS,NEWSTATE),asserta(time_stamp(NTS)),
            means_ends_tutor(NEWSTATE,GOAL,OPLIST,GOALSTATE,STACK,GOALSTACK).


    do_intro :-
         intro(T),nl(2), write(T), nl(2), ! .
    do_intro .


    /****  PROBLEM-DEFINITION ERROR CHECKING  ****/

    check_obvious_errors :-
         setof([M,A],obvious_error(M,A),MAL), !, writepairlist(MAL) .

    obvious_error("precondition fact missing for operator ",O) :-
         recommended(D,O), not precondition(O,L) .
    obvious_error("deletepostcondition fact missing for operator ",O) :-
         recommended(D,O), not get_deletepostcondition(O,S,L) .
    obvious_error("addpostcondition fact missing for operator ",O) :-
         recommended(D,O), not get_addpostcondition(O,S,L) .
    obvious_error("$"recommended$" fact missing for operator ",O) :-
         precondition(O,L), not recommended(D,O) .
    obvious_error("$"recommended$" fact missing for operator ",O) :-
         get_deletepostcondition(O,S,L), not recommended(D,O) .
    obvious_error("$"recommended$" fact missing for operator ",O) :-
         get_addpostcondition(O,S,L), not recommended(D,O) .


    /****  HANDLING OF RANDOMNESS  ****/

    do_randsubst(O,S,NS) :-
         randsubst(O,RL), !, do_randsubst2(RL,S,NS) .
    do_randsubst(O,S,S) .


    do_randsubst2([],S,S) .
    + do_randsubst2(--F-|L-,S,NS) :-
            member(F,S),delete(F,S,S2),!,do_randsubst2(L,S2,NS).
    + do_randsubst2(--F,M-|L-,S,NS) :-
            member(F,S),delete(F,S,S2),nl,write(M),nl,do_randsubst2(L,S2,NS).
    do_randsubst2([[F,NF,P]|L],S,NS) :-
         rand(1000,K), P1000 is P*1000, K=<P1000, changestate(F,NF,S,S2), !,
            do_randsubst2(L,S2,NS) .
    do_randsubst2([[F,NF,P,M]|L],S,NS) :-
         rand(1000,K), P1000 is P*1000, K=<P1000, changestate(F,NF,S,S2), !,
            write(M), nl, do_randsubst2(L,S2,NS) .
    do_randsubst2([C|L],S,NS) :-
         do_randsubst2(L,S,NS) .


    changestate(none,NF,S,[NF|S]) :-
         !, not member(NF,S), write("Random change made: fact "),
            writefact(NF,state), write(" added."), nl, ! .
    changestate(F,none,S,S2) :-
         !, member(F,S), delete(F,S,S2), write("Random change made: fact "),
            writefact(F,state), write(" removed."), nl, ! .
    changestate(F,NF,S,[NF|S3]) :-
```

54

```prolog
       !, member(F,S), delete(F,S,S3), write("Random change made: fact "),
          writefact(NF,state), write(" added, and"), nl, write("fact "),
          writefact(F,state), write(" removed."), nl, ! .

permutation([],[]) :-
       ! .
permutation(L,[I|PL]) :-
       randitem(L,I), delete(I,L,L2), permutation(L2,PL) .

randitem(L,I) :-
       length(L,N), rand(N,KM1), K is KM1+1, item(K,L,I) .

rand(N,K) :-
       KR is random*N, K is int(KR) .

item(K,[],I) :-
       !, fail .
item(K,[X|L],X) :-
       K=<1, ! .
item(K,[X|L],Y) :-
       KM1 is K-1, item(KM1,L,Y) .

/****  TUTORING RULES  ****/

* check_with_student(O,S,D,NO) :-
       write(***************************************), nl,
          write("The following facts are now true:"), nl, writelist(S,state),
          write("."), nl, write("What operator do you choose? "), nl,
          niceread(AO2),!,check_time_stamp, space_parse(AO2,O3),!,
          handle_student_op(O3,O,S,D,NO) .

handle_student_op(O,O,S,D,O) :-
       !, write("OK."), nl .
handle_student_op('',O,S,D,NO) :-
       !, check_with_student(O,S,D,NO) .
handle_student_op(O2,O,S,D,NO) :-
       helpword(O2), !, op_list(OL), permutation(OL,POL),
          write("The possible operators are: "), writelist(POL,op),
          write("."), nl, write("Your objectives are: "), top_goal(G),
          writelist(G,state), write("."), nl,!, check_with_student(O,S,D,NO) .
handle_student_op(O2,O,S,D,O) :-
       op_list(OL), not singlemember(O2,OL), fixspell(O2,O), !,
          write("I assume you mean "), write(O), write("."), nl .
handle_student_op(O2,O,S,D,NO) :-
       op_list(OL), not singlemember(O2,OL), backtracking_member(O3,OL),
          fixspell(O2,O3), !, write("I assume you mean "), write(O3),
          write("."), nl, handle_student_op(O3,O,S,D,NO) .
handle_student_op(O2,O,S,D,NO) :-
       op_list(OL), not singlemember(O2,OL), !,
          write("Not a valid operator--please choose one of: "),
          permutation(OL,POL), writelist(POL,op), write("."), nl,
          check_with_student(O,S,D,NO) .
handle_student_op(O2,O,S,D,NO) :-
       precondition(O2,PO2), difference(PO2,S,D2), not D2=[], !,
          write("That operator requires that "), writelist(D2,precond),
```

```
            write("."), nl, check_with_student(O,S,D,NO) .
handle_student_op(O2,O,S,D,NO) :-
      apply_op(O2,S,S), write("That will not affect anything."), nl,
         check_with_student(O,S,D,NO) .
handle_student_op(O2,O,S,D,NO) :-
      apply_op(O2,S,S2), top_goal(G), not once_means_ends(S2,G,OL2,GS2), !,
         write("You cannot ever succeed if you do that."), nl,
         check_with_student(O,S,D,NO) .
handle_student_op(O2,O,S,D,O2) :-
      top_goal(G), apply_op(O,S,S3), apply_op(O2,S,S2),
         compare_solutions(S3,G,OL3,GS3,S2,G,OL2,GS2),
         subsequence([O|OL3],OL2), !, apply_ops([O|OL3],S,SL,GS4),
         elimdups(SL,ESL), asserta(mainline_states(ESL,O2,S,O)),
         write("That does not seem immediately helpful, but I will try it."
            ), nl .
handle_student_op(O2,O,S,D,O2) :-
      nopref(O2,O), !, write("OK."), nl .
handle_student_op(O2,O,S,D,O2) :-
      nopref(O,O2), !, write("OK."), nl .
handle_student_op(O2,O,S,D,O2) :-
      top_goal(G), once_means_ends(S,G,OL,FS), not member(O2,OL), !,
         write("I will try it, but it is not recommended or needed for the$
            $ problem."), nl .
handle_student_op(O2,O,S,D,O2) :-
       top_goal(G), difference(G,S,D2), all_achievable(S,D2),
         applicable_op(D2,O3), precondition(O3,PL),
         least_common_op(S,G,O,O2,PL,GROOT), !,
         write("I will try it, but it is not recommended first when "),
         difference(GROOT,S,D5), delete_uncreatable(D5,D6),
         permutation(D6,D7), writelist(D7,precond), write("."), nl .
handle_student_op(O2,O,S,D,O2) :-
      write("Not the operator I would choose, but let us try it."), nl, ! .


/****   INTERMEDIATE PREDICATES USED BY THE TUTOR   ****/

least_common_op(S,G,O,O2,G2,G) :-
      once_means_ends(S,G2,OL,NS), least_common_op2(O,O2,OL) .
least_common_op(S,G,O,O2,G2,DROOT) :-
      difference(G2,S,D), all_achievable(S,D), applicable_op(D,O3),
         precondition(O3,G3), least_common_op(S,G2,O,O2,G3,DROOT), ! .

least_common_op2(O,O2,OL) :-
      not member(O,OL), ! .
least_common_op2(O,O2,OL) :-
      not member(O2,OL), ! .

compare_solutions(S3,G,OL3,GS3,S2,G,OL2,GS2) :-
      once_means_ends(S3,G,OL3,GS3), once_means_ends(S2,G,OL2,GS2), ! .

cache_states(S,G,[],GS) :-
      ! .
cache_states(S,G,OL,GS) :-
      cached(S,G,OL,GS), ! .
cache_states(S,G,OL,GS) :-
      cached(S2,G2,OL2,GS2), check_permutation(S,S2),
```

```prolog
            check_permutation(G,G2), ! .
cache_states(S,G,[O|OL],GS) :-
      asserta(cached(S,G,[O|OL],GS)), apply_op(O,S,NS),
            cache_states(NS,G,OL,GS), ! .


apply_ops([],S,[S],S) :-
      ! .
apply_ops([O|OL],S,[S|SL],NS) :-
      apply_op(O,S,S2), apply_ops(OL,S2,SL,NS) .

apply_op(O,S,NS) :-
      get_deletepostcondition(O,S,DP), deleteitems(DP,S,S2),
            get_addpostcondition(O,S,AP), union(AP,S2,NS), ! .

helpword(help) .
helpword(h) .
helpword(huh) .

check_mainline_return(S) :-
      mainline_states(SL,O,OS,BO), check_mainline_return2(S,SL,O,OS,BO) .
check_mainline_return(S) .

check_mainline_return2(S,[S2|SL],O,OS,BO) :-
      permutemember(S,[S2]), !,
      write("You are returning to a previous state."), nl .
check_mainline_return2(S,SL,O,OS,BO) :-
      permutemember(S,SL), !,
         write("Do you see now that your choice of the "), write(O),
         write(" action in the state with the facts ["),
         writelist(OS,state), write("] was not the best choice; the "),
         write(BO), write(" action would have been better."), nl,
         del_statement(mainline_states(SL,O,OS,BO)) .

higher_goal_achieved(GL,S) :-
      higher_goal_achieved2(GL,S) .

higher_goal_achieved2([],S) :-
      !, fail .
higher_goal_achieved2([G|GL],S) :-
      difference(G,S,[]), ! .
higher_goal_achieved2([G|GL],S) :-
      higher_goal_achieved2(GL,S) .


/****  NATURAL LANGUAGE OUTPUT HANDLER  ****/

writelist([],R) :-
      ! .
writelist([X],R) :-
      !, writefact(X,R) .
writelist([X,Y],R) :-
      !, writefact(X,R), write(" and "), writefact(Y,R) .
writelist(L,R) :-
      writelist2(L,R) .
```

```prolog
writelist2([X],R) :-
    !, write("and "), writefact(X,R) .
writelist2([X|L],R) :-
    writefact(X,R), write(", "), writelist2(L,R) .


writefact(F,state) :-
    atom(F), write("it is "), write(F), ! .
writefact(not F,state) :-
    atom(F), !, write("it is not "), write(F), ! .
writefact(not F,state) :-
    F=..[P,X], atom(X), !, write(X), is_form(X,IX), write(IX),
      write("not "), write(P), ! .
writefact(not F,state) :-
    F=..[P,X], !, writefact(X), is_form(X,IX), write(IX), write("not "),
      write(P), ! .
writefact(not F,state) :-
    F=..[P,X,Y], !, write(P), write(" not "), write(X), write(" "),
      write(Y), ! .
writefact(F,state) :-
    F=..[P,X], atom(X), !, write(X), is_form(X,IX), write(IX), write(P),
      ! .
writefact(F,state) :-
    F=..[P,X], !, writefact(X,state), is_form(X,IX), write(IX), write(P),
      ! .
writefact(F,state) :-
 F=..[P,X,Y], !, write(P), write(" "), write(X), write(" "), write(Y), ! .
writefact(F,precond) :-
    atom(F), write("it must be "), write(F), ! .
writefact(not F,precond) :-
    atom(F), !, write("it must not be "), write(F), ! .
writefact(not F,precond) :-
    F=..[P,X], atom(X), !, write(X), write(" must not be "), write(P), ! .
writefact(not F,precond) :-
    F=..[P,X], !, writefact(X,state), write(" must not be "), write(P),
      ! .
writefact(not F,precond) :-
    F=..[P,X,Y], !, write(P), write(" must not be "), write(X),
      write(" "), write(Y), ! .
writefact(F,precond) :-
    F=..[P,X], atom(X), !, write(X), write(" must be "), write(P), ! .
writefact(F,precond) :-
    F=..[P,X], !, writefact(X,state), write(" must be "), write(P), ! .
writefact(F,precond) :-
    F=..[P,X,Y], write(P), write(" must be "), write(X), write(" "),
      write(Y), ! .
writefact(F,op) :-
    F=..[P,A], write(P), write(" "), write(A), ! .
writefact(F,op) :-
    F=..[P,A,B], write(P), write(" "), write(A), write(" "), write(B), ! .
writefact(F,op) :-
    write(F), ! .
writefact(F,R) :-
    write(F) .
```

```
/*A simple heuristic is used for plurals:   the thing before the "is"*/
/*is plural if it ends in "s".*/
is_form(X," is ") :-
    not atom(X), ! .
is_form(X," are ") :-
    name(X,NX), last(NX,115), ! .
is_form(X," is ") .


/****   ORIGINAL MEANS-ENDS PROGRAM   ****/

once_means_ends(STATE,GOAL,OPLIST,GOALSTATE) :-
    means_ends(STATE,GOAL,OPLIST,GOALSTATE),
        cache_states(STATE,GOAL,OPLIST,GOALSTATE), ! .

means_ends(STATE,GOAL,OPLIST,GOALSTATE) :-
    means_ends2(STATE,GOAL,OPLIST,GOALSTATE,[]), writedebug7 .

means_ends2(STATE,GOAL,OPLIST,GOALSTATE,STACK) :-
    cached(STATE2,GOAL2,OPLIST,GOALSTATE), check_permutation(GOAL,GOAL2),
        check_permutation(STATE,STATE2), !, writedebug6(STACK), ! .
means_ends2(STATE,GOAL,OPLIST,GOALSTATE,STACK) :-
    member([STATE,GOAL],STACK), !, writedebug4(STATE,GOAL,STACK), fail .
means_ends2(STATE,GOAL,[],STATE,STACK) :-
    difference(GOAL,STATE,[]), ! .
means_ends2(STATE,GOAL,OPLIST,GOALSTATE,STACK) :-
    difference(GOAL,STATE,D), applicable_op(D,OPERATOR),
        precondition(OPERATOR,PRELIST), all_achievable(STATE,PRELIST),
        writedebug1(D,OPERATOR,STACK),
        means_ends2(STATE,PRELIST,PREOPLIST,PRESTATE,[[STATE,GOAL]|STACK]),
        writedebug2(PRESTATE,D,OPERATOR,STACK),
        get_deletepostcondition(OPERATOR,PRESTATE,DELETEPOSTLIST),
        deleteitems(DELETEPOSTLIST,PRESTATE,PRESTATE2),
        get_addpostcondition(OPERATOR,PRESTATE,ADDPOSTLIST),
        union(ADDPOSTLIST,PRESTATE2,POSTLIST),
        means_ends2(POSTLIST,GOAL,POSTOPLIST,GOALSTATE,[[STATE,GOAL]|STACK
            ]), writedebug3(GOALSTATE,OPERATOR,STACK),
        append(PREOPLIST,[OPERATOR|POSTOPLIST],OPLIST) .
means_ends2(STATE,GOAL,OPLIST,GOALSTATE,STACK) :-
    writedebug5(STATE,GOAL,STACK), !, fail .

/****   DEBUGGING TOOLS   ****/

writedebug1(D,O,STACK) :-
    not debugflag, ! .
writedebug1(D,O,STACK) :-
    length(STACK,NM1), N is NM1+1, write(">>Operator "), write(O),
        write(" suggested at level "), write(N), nl,
        write("to achieve difference of ["), writelist(D,state),
        write("]"), nl, ! .

writedebug2(S,D,O,STACK) :-
    not debugflag, ! .
writedebug2(S,D,O,STACK) :-
    length(STACK,NM1), N is NM1+1, write(">>Operator "), write(O),
        write(" applied at level "), write(N), nl,
        write("to reduce difference of ["), writelist(D,state), write("]"),
```

```
          not debugflag, ! .
writedebug3(S,O,STACK) :-
       length(STACK,NM1), N is NM1+1, write(">>Level "), write(N),
         write(" terminated at state in which "), writelist(S,state), nl, ! .

writedebug4(S,G,STACK) :-
       not debugflag, ! .
writedebug4(S,G,STACK) :-
       write(">>>>Reasoning avoided an infinite loop at level "),
         length(STACK,NM1), N is NM1+1, write(N),
         write(" where problem was identical to that at level "),
         index([S,G],STACK,I), write(I), nl, ! .

writedebug5(STATE,GOAL,STACK) :-
       not debugflag, ! .
writedebug5(STATE,GOAL,STACK) :-
       write(">>>>Unsolvable problem at level "), length(STACK,NM1),
         N is NM1+1, write(N), nl, write("for state "),
         writelist(STATE,state), nl, write("and goal "),
         writelist(GOAL,state), nl, ! .

writedebug6(STACK) :-
       not debugflag, ! .
writedebug6(STACK) :-
       . write(">>>>Previously computed solution used at level "),
         length(STACK,NM1), N is NM1+1, write(N), nl, ! .

writedebug7 :-
       not debugflag, ! .
writedebug7 :-
       nl, ! .

writedebug8(OP) :-
       not debugflag, ! .
writedebug8(OP) :-
       write("The tutor prefers operator "),
         write(OP), nl, ! .

/****  UTILITY FUNCTIONS  ****/

delete_uncreatable([],[]) .
delete_uncreatable([X|L],M) :-
       uncreatable(X), !, delete_uncreatable(L,M) .
delete_uncreatable([X|L],[X|M]) :-
       delete_uncreatable(L,M) .

all_achievable(S,G) :-
       difference(G,S,D), not unachievable_member(D) .

unachievable_member(D) :-
       backtracking_member(F,D), uncreatable(F) .



uncreatable(F) :-
       precondition(O,L), backtracking_member(F,L), not in_postcondition(F) .
```

```prolog
in_postcondition(not F) :-
    any_deletepostcondition(0,DPL), member(F,DPL), ! .
in_postcondition(not F) :-
    randsubst(0,RSL), member([F,X,Y,Z],RSL), ! .
in_postcondition(not F) :-
    randsubst(0,RSL), member([F,X,Y],RSL), ! .
in_postcondition(F) :-
    not F=..[not,P], any_addpostcondition(0,APL), member(F,APL), ! .
in_postcondition(F) :-
    not F=..[not,P], randsubst(0,RSL), member([X,F,Y,Z],RSL), ! .
in_postcondition(F) :-
    not F=..[not,P], randsubst(0,RSL), member([X,F,Y],RSL), ! .


any_deletepostcondition(0,L) :-
    deletepostcondition(0,L) .


any_addpostcondition(0,L) :-
    addpostcondition(0,L) .


get_deletepostcondition(0,S,L) :-
    deletepostcondition_args(4), deletepostcondition(0,C,L,M),
        factsubset(C,S), ! .
get_deletepostcondition(0,S,L) :-
    , deletepostcondition_args(3), deletepostcondition(0,C,L),
        factsubset(C,S), ! .
get_deletepostcondition(0,S,L) :-
    deletepostcondition_args(2), deletepostcondition(0,L) .


get_addpostcondition(0,S,L) :-
    addpostcondition_args(4), addpostcondition(0,C,L,M), factsubset(C,S),
        ! .
get_addpostcondition(0,S,L) :-
    addpostcondition_args(3), addpostcondition(0,C,L), factsubset(C,S),
        ! .
get_addpostcondition(0,S,L) :-
    addpostcondition_args(2), addpostcondition(0,L) .


print_optional_message_d(0,S) :-
    deletepostcondition_args(4), deletepostcondition(0,C,L,M),
        factsubset(C,S), write(M), nl, ! .
print_optional_message_d(0,S) :-
    ! .


print_optional_message_a(0,S) :-
    addpostcondition_args(4), addpostcondition(0,C,L,M), factsubset(C,S),
        write(M), nl, ! .
print_optional_message_a(0,S) :-
    ! .


applicable_op(D,0) :-
    recommended(D2,0), subset(D2,D) .


fixspell(W1,W2) :-
    atom(W1), atom(W2), !, name(W1,AW1), fixspell2(AW1,AW2),
```

```prolog
            name(W2,AW2) .
fixspell(W1,W2) :-
     structure(W1,list,[P1|L]), structure(W2,list,[P2|L]), not P1=P2,
     !,fixspell(P1,P2) .
fixspell(W1,W2) :-
     structure(W1,list,[P,Q1|L]), structure(W2,list,[P,Q2|L]), not Q1=Q2,
      !, fixspell(Q1,Q2) .
fixspell(W1,W2) :-
     structure(W1,list,[P,Q,R1|L]), structure(W2,list,[P,Q,R2|L]),
      not R1=R2, !, fixspell(R1,R2) .


fixspell2(AW,AW2) :-
     deleteone(X,AW,AW2) .
fixspell2(AW,AW2) :-
     deleteone(X,AW2,AW) .
fixspell2(AW,AW2) :-
     transpose(AW,AW2) .

transpose([X,Y|L],[Y,X|L]) .
transpose([X|L],[X|M]) :-
     transpose(L,M) .



deleteone(X,[X|L],L) .
deleteone(X,[Y|L],[Y|M]) :-
     deleteone(X,L,M) .


difference([],S,[]) .
difference([not P|G],S,G2) :-
     not singlemember(P,S), !, difference(G,S,G2) .
difference([P|G],S,G2) :-
     singlemember(P,S), !, difference(G,S,G2) .
difference([P|G],S,[P|G2]) :-
     difference(G,S,G2) .

subset([],L) .
subset([X|L],L2) :-
     singlemember(X,L2), subset(L,L2) .

factsubset([],L) .
factsubset([not P|L],L2) :-
     not singlemember(P,L2), !, factsubset(L,L2) .
factsubset([not P|L],L2) :-
     !, fail .
factsubset([P|L],L2) :-
     singlemember(P,L2), factsubset(L,L2) .

member(X,L) :-
     singlemember(X,L) .

singlemember(X,[X|L]) :-
     ! .
singlemember(X,[Y|L]) :-
     singlemember(X,L) .
```

```prolog
append([],L,L) .
append([X|L],L2,[X|L3]) :-
    append(L,L2,L3) .

union([],L,L) .
union([X|L1],L2,L3) :-
    singlemember(X,L2), !, union(L1,L2,L3) .
union([X|L1],L2,[X|L3]) :-
    union(L1,L2,L3) .

deleteitems([],L,L) .
deleteitems([X|L],L2,L3) :-
    delete(X,L2,L4), deleteitems(L,L4,L3) .

delete(X,[],[]) .
delete(X,[X|L],M) :-
    !, delete(X,L,M) .
delete(X,[Y|L],[Y|M]) :-
    delete(X,L,M) .

check_permutation(L,M) :-
    subset(L,M), subset(M,L), ! .

subsequence([],L) :-
    ! .
subsequence([X|L],[X|M]) :-
    !, subsequence(L,M) .
subsequence(L,[X|M]) :-
    subsequence(L,M) .

permutemember(X,[X|L]) :-
    ! .
permutemember(X,[Y|L]) :-
    subset(X,Y), subset(Y,X), ! .
permutemember(X,[Y|L]) :-
    permutemember(X,L) .

last([X],X) .
last([X|L],Y) :-
    last(L,Y) .

elimdups([],[]) .
elimdups([X|L],M) :-
    singlemember(X,L), !, elimdups(L,M) .
elimdups([X|L],[X|M]) :-
    elimdups(L,M) .

uniqueassert(Q) :-
    del_statement(Q), !, add_statement(Q) .
uniqueassert(Q) :-
    add_statement(Q) .

backtracking_member(X,[X|L]) .
backtracking_member(X,[Y|L]) :-
    backtracking_member(X,L) .
```

```
/**** INPUT/OUTPUT HANDLER  ****/

space_parse([32|AS1],S1) :-
     space_parse(AS1,S1), ! .
space_parse(AS1,S1) :-
     not member(32,AS1), remove_ugly_chars(AS1,AS2), name(S1,AS2), ! .
space_parse(AS1,S2) :-
     remove_ugly_chars(AS1,AS2), append(L1,[32|L2],AS2), name(N1,L1),
        parse_args(L2,N2), S2=..[N1|N2] .

parse_args([],[]) :-
     ! .
parse_args([32|AL],L) :-
     parse_args(AL,L), ! .
parse_args(AL,[L]) :-
     not member(32,AL), name(L,AL), ! .
parse_args(AL,[N1|N2]) :-
     append(L1,[32|L2],AL), name(N1,L1), parse_args(L2,N2), ! .

remove_ugly_chars([],[]) .
remove_ugly_chars([X|L],M) :-
      X<65, not X=32, !, remove_ugly_chars(L,M) .
remove_ugly_chars([X|L],[X|M]) :-
     remove_ugly_chars(L,M) .

niceread(L) :-
     checkretract(readbuff(L2)), asserta(readbuff([])), niceread2(L),! .

niceread2(L) :-
     get0(C), niceread3(C,L) .

niceread3(10,L) :-
     !, readbuff(L2), reverse(L2,L) .
niceread3(C,L) :-
     readbuff(L3), retract(readbuff(L3)), asserta(readbuff([C|L3])),
        niceread2(L) .

+ check_time_stamp :-
        read_fact_file(TS,S),time_stamp(TS1),asserta(ts_violation),
        !,compare(=,TS,TS1),retract(ts_violation).

+ update_fact_file(NTS,NS) :-
        set_channel(outfile(outf),name="facts.pro"),
        set_channel(outfile(outf),buffer=1000),
        set_output(outfile(outf)),
        write(NTS),nl,write(NS),
        close_output(outfile(outf)).

+ read_fact_file(TS,S) :-
        set_channel(infile(dummy),name="/work/salgado/meunrep.pro"),
        set_channel(infile(inf),name="facts.pro"),
        set_input(infile(inf)),
        skip_unread_input,
```

```
        read(TS),read(S),
        close_input(infile(inf)).

checkretract(S) :-
    call(S), retract(S), ! .
checkretract(S) .

reverse(L,R) :-
    reverse2(L,[],R) .

reverse2([],L,L) :-
    ! .
reverse2([X|L],R,S) :-
    reverse2(L,[X|R],S) .

index(X,[X|L],1) :- !.
index(X,[Y|L],N) :- index(X,L,Nm1), N is Nm1+1.

endmod /* metutor11 */  .
```

LIST OF REFERENCES

1. Department of Defense, Office of the Assistant Secretary of Defense for Force Management and Personnel, *Military Manpower Training Report: FY 1988*, Government Printing Office, Washington, DC, 1987.

2. Cook, P., Kane, T.P., and McQuie, R., *Advanced Technology Manpower Forecasting*, BKD TR-3-235, B-K Dynamics, Rockville, Maryland, 1977.

3. Barr, A. and Feigenbaum, E. A., *The Handbook of Artificial Intelligence*, v. 2, William Kaufmann, 1982.

4. Kearsley, G. P., *Artificial Intelligence & Instruction*, Addison-Wesley, 1987.

5. Steinberg, E. R., *Teaching Computers to Teach*, Lawrence Erlbaum Associates, 1984.

6. Fletcher, J. D., *Intelligent Training Systems in the Military*, in Andriole, S. J. and Hopple, G. W., *Defense Applications of Artificial Intelligence*, Lexington Books, 1988.

7. Sleeman, D. and Brown, J. S., *Intelligent Tutoring Systems*, Academic Press, 1982.

8. Wenger, E., *Artificial Intelligence and Tutoring Systems*, Morgan Kaufmann Publishers, 1987.

9. Campbell, D. S., *An Intelligent Computer-Assisted Instruction System for Cardiopulmonary Resuscitation*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1988.

10. Kim, T. W., *A Computer-Aided Instruction Program for Teaching the TOPS20-MM Facility on the DDN*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1988.

11. Rowe, N. C., *Artificial Intelligence Through Prolog*, Prentice-Hall, 1988.

12. Department of the Navy, Office of the Chief of Naval Operations, *Replenishment at Sea, NWP 14 (Rev. D)*, January 1989.

13. *Prolog-86 (tm)*, Version 1.12, Copyright 1983, Micro-AI, Moraga, California, 1984.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, CA 9394-5000	2
4.	Professor Neil C. Rowe, Code 52Rp Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	2
5.	Jefatura de Operaciones Navales Emabassy of Ecuador Office of the Naval Attache 2535 - 15th St. NW Washington, DC 20009	1
6.	Dirección General de Educación de la Armada Emabassy of Ecuador Office of the Naval Attache 2535 - 15th St. NW Washington, DC 20009	2
7.	Academia de Guerra Naval Emabassy of Ecuador Office of the Naval Attache 2535 - 15th St. NW Washington, DC 20009	1
8.	Dirección de Informática de la Armada Emabassy of Ecuador Office of the Naval Attache 2535 - 15th St. NW Washington, DC 20009	1
9.	TNFG-UN Patricio J. Salgado Z. Av. Edmundo Carvajal #718 Quito, Ecuador (South America)	2