

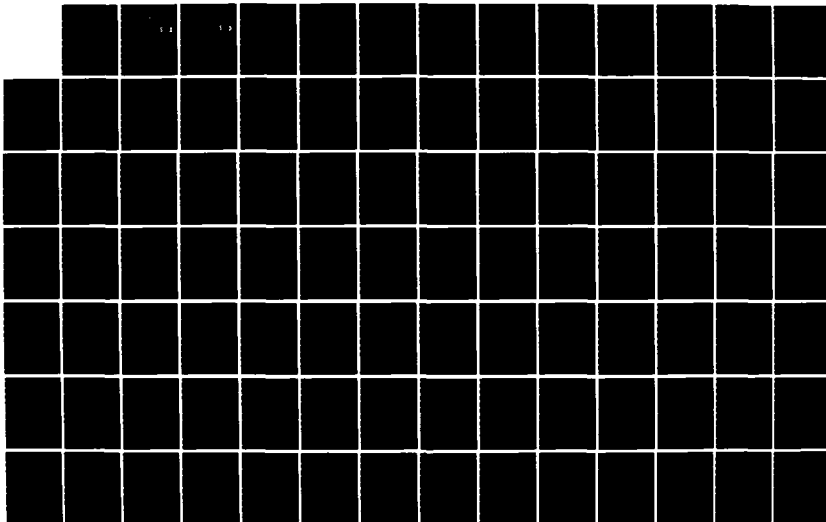
AD-A163 956

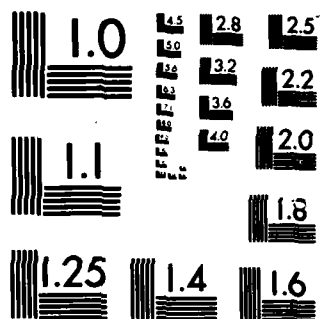
AUTONOMOUS VEHICLE MISSION PLANNING USING AI
(ARTIFICIAL INTELLIGENCE) TE. (U) AIR FORCE INST OF
TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.
S E STOCKBRIDGE DEC 85 AFIT/GE/ENG/85D-45 F/G 6/4

1/3

UNCLASSIFIED

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD-A163 956



DTIC
ELECTE
FEB 13 1986

S

D

AUTONOMOUS VEHICLE MISSION PLANNING

USING AI TECHNIQUES

THESIS

Samuel E. Stockbridge
Captain, USAF

AFIT/GE/ENG/85D-45

DISTRIBUTION STATEMENT A

Approved for public release
Distribution Unlimited

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

86 2 13 015

MMC FILE COPY

AFIT/GE/ENG/85D-45

1

DTIC
ELECTE
FEB 13 1986
S D D

AUTONOMOUS VEHICLE MISSION PLANNING

USING AI TECHNIQUES

THESIS

Samuel E. Stockbridge
Captain, USAF

AFIT/GE/ENG/85D-45

Approved for public release; distribution unlimited

AFIT/GE/ENG/85D-45

AUTONOMOUS VEHICLE MISSION PLANNING USING AI TECHNIQUES

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Electrical Engineering

Samuel E. Stockbridge, B.S.
Captain, USAF

December 1985



Accession For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail. and/or Special
A-1	

Approved for public release; distribution unlimited

Preface

The purpose of this thesis was to investigate the critical components of an autonomous vehicle's "intelligence". This thesis gives particular emphasis to the desirable attributes of an operating system and mission planner, taking concepts from the field of cognitive psychology and natural language text understanding and incorporating them into the vehicle's planning architecture.

The argument given here is that in order for an autonomous vehicle to be truly intelligent, and hence truly autonomous, it must have the ability to understand its environment as well as the ability to plan in it. Therefore, a blackboard control architecture was adopted for the operating system in order to provide flexibility. The mission planner architecture, on the other hand, was based on a production system using meta-knowledge, that is, knowledge about the planning process, in order to construct plans. The planner is the plan generator, while the operating system is a plan projector and goal detector.

Although the blackboard control architecture is well known, I feel that the particular operating system architecture combined with the planner architecture implemented here provides a more powerful and flexible mechanism for overall control of the autonomous vehicle.

Samuel E. Stockbridge

Table of Contents

	Page
Preface	ii
List of Figures	v
Abstract	vi
I. Introduction	1
Background	1
Problem	2
Scope	6
Assumptions	6
Summary of Current Knowledge	7
Approach	11
Equipment Requirements	12
II. Autonomous Vehicle Operating System	13
Introduction	13
Autonomous Vehicle Model	16
Intelligent Operating System Tasks	18
III. Planning	22
Introduction	22
Hierarchical Planning	23
Nonhierarchical Planning	28
Metapanning	30
System Organization	34
IV. Operating System Design	37
Introduction	37
The ROSS Language	39
The Program Structure	42
The World Model	55
V. Planning System Design	59
Introduction	59
Conceptual Dependency	61
Micro-PAM	65
Program Structure	72

VI.	Integration and Testing	81
	Introduction	81
	Integration	82
	Testing	82
VII.	Summary, Conclusions, and Recommendations . . .	113
	Summary and Conclusions	113
	Recommendations	116
	Appendix: Test Run Transcript	119
	Bibliography	198
	Vita	200

List of Figures

Figure	Page
1. Plan for Days Activities	24
2. Non-linear Planning	27
3. Meta-Planning Process	31
4. Program Architecture	43
5. Communication Among Main Processor Actors . . .	43
6. Communication Among Sensor Actors	50
7. Blackboard Partitions	52
8. World Model of Autonomous Vehicle's Environment	56
9. Base to Workbench Route	56
10. Truth Maintenance	74
11. Autonomous Vehicle's Environment	84

Abstract

This study investigates a software architecture for autonomous vehicle control. The autonomous vehicle's planning ability is divided into operating system functions and mission planning system functions. The blackboard control architecture is adopted for the operating system design with implementation using the ROSS programming language.

The planning system incorporates elements of a planner and understander by declaratively encoding meta-knowledge, or knowledge about the planning process. By separating the knowledge about how to plan from the specific domain knowledge, an understander can use this knowledge about how plans are constructed, in combination with the specific domain knowledge, in the understanding process. Likewise, a planner can use this same knowledge in the planning process. Thus, a great deal of flexibility is attained by dividing the knowledge base into meta-rules and domain specific rules.

The planning system constructs an agenda of scripts which directs the control flow in the operating system. The operating system is given the additional duties of goal detector and plan projector in order to simulate the plan steps proposed by the planner. Hence, the operating system detects any errors in the

plan and terms these errors in the form of goals the planner can understand.

The implementation demonstrates the benefits of using meta-planning concepts combined with a blackboard control architecture to provide an autonomous vehicle with a more flexible and powerful planning capability.

AUTONOMOUS VEHICLE MISSION PLANNING USING AI TECHNIQUES

I. Introduction

Background

The ability to maintain and service aircraft in a Nuclear-Biological-Chemical (NBC) contaminated environment has long been of interest to the Air Force. Research efforts by the Air Force have been concentrated in reducing the risk to ground crews while still maintaining the vital function of aircraft maintenance. One approach investigated by the Air Force involves the use of an autonomous vehicle (such as a robot) to perform the tasks of aircraft maintenance (3; 12). To accomplish the job, the autonomous vehicle must have the ability to:

1. Plan actions and prioritize assigned tasks
2. Plan a path to the appropriate aircraft
3. Handle unforeseen problems
4. Accomplish the required maintenance
5. Return to the work station

A previous AFIT thesis effort focused on the problem of planning an optimum path and navigating the autonomous vehicle to its destination (10). The effort resulted in a software algorithm that can determine the shortest path to a goal around stationary objects in an aircraft hangar. The algorithm requires

an accurate representation of the vehicle's environment along with predetermined information such as the current location of the vehicle and the destination. Given this information, the algorithm can simulate the movements of the vehicle through an aircraft hangar; however, it cannot avoid dynamic objects (objects in motion) that may cross its path. Nevertheless, the algorithm provides an excellent foundation for further research.

Problem

The ability to plan actions and prioritize tasks is crucial to the development of an autonomous vehicle capable of operating with minimal human intervention. This research effort will concentrate on developing a software algorithm to perform the mission planning function while using available on-board resources effectively.

There are many tasks that a human ground crewman performs while striving to accomplish his ultimate goal. For example, if his ultimate goal is to repair a jet engine, he must first have the necessary tools with him. If he does not have the tools, he must construct a plan to obtain them. Once he has obtained the tools, he must next construct a plan to get to the appropriate aircraft so that he can accomplish his ultimate goal. Humans do not consciously construct plans for these tasks because they have the ability to learn and to use their prior experiences in similar situations. An autonomous vehicle, on the other hand, must have all this detailed knowledge available to it. It must know what the preconditions are for accomplishing tasks, and it must perform them in the proper sequence. In the previous

example, if an autonomous vehicle was told to repair the engine on a jet, it must know that the preconditions for accomplishing this task are to be at the jet, and to have the necessary tools. However, simply knowing these preconditions is not enough; it must also know that the tools must be obtained first. Having this knowledge available to the vehicle is essential to the mission planning capability of the autonomous vehicle.

An autonomous vehicle capable of operating in its environment independent from human intervention must have the ability to use its resources effectively. A human has the ability to process a myriad of information in parallel and utilize this information in performing everyday activities. The actual processing of this information may be done in different parts of the brain; however, it is made available to other parts if needed. Likewise, an autonomous vehicle must process a host of information from various sources and use this information to perform its prescribed mission. The type of information the autonomous vehicle might process would include: sensor data, route planning data, mission planning data, as well as information on the vehicle's fuel status and maintenance requirements. Assimilating and controlling the flow of this information is the job of the operating system, and it is crucial to the mission of the autonomous vehicle.

In a multiprocessor autonomous vehicle, direct communication between processors may not be feasible due to design constraints on available area for bus paths. Furthermore, the complexity of direct processor to processor communication may make future upgrade of the system extremely difficult. Instead, communication

between processors can be accomplished through a shared common memory or blackboard. By using the blackboard approach, system information can be made available to any processor needing it. The difficulty with this approach, however, is the requirement to control accesses to the blackboard. One purpose of this project is to simulate the processing environment present in an autonomous vehicle in an effort to identify a suitable software architecture that would allow an autonomous vehicle to perform its mission effectively.

Unlike an operating system for a regular computer, the operating system in an autonomous vehicle must be able to handle unforeseen situations. For example, if a certain plan is being carried out that makes use of the vehicle's sensory capability (such as sonars) on the left side of its body, and all sensors fail on the left side, then the operating system may construct a plan to use other available sensors, or to use the sensors on the right side of its body. To achieve this, the operating system must be able to infer goals. If in the example above, the vehicle's available fuel became critically low, it would be desirable to give maximum attention to planning a route to a refueling station. Assuming the vehicle were stationary at the time, it would be unwise to give equal bus time to the other processors. Instead, effort should go into correcting the situation and perhaps even shutting down unnecessary processes. The detection of such a goal might be the job of an execution monitor who constantly checks on the condition of the vehicle. If a low fuel state is detected, this information would be passed to a

projector to determine if the vehicle's current goal could be achieved and still allow sufficient fuel to reach its refueling station. If the current goal could not be achieved, it should be abandoned and a new goal of refueling should be established. In effect, the operating system's flow of control has been altered; new priorities have been established and previous policies the system may have had, such as allowing equal bus time, are abandoned. To accomplish this, it will be necessary to integrate the mission planning mechanism with the overall operating system architecture to allow these types of goal inferences.

An independent but important function of the mission planning problem is the path planning problem. Planning an optimum route to a destination can be accomplished if the path planner has at its disposal an accurate representation of its environment. However, while executing the planned route the vehicle may encounter dynamic objects or objects that it had no knowledge of. One strategy might be to simply stop and wait for a moving object to cross the path; however, this strategy may not work in all situations. The moving object may be on a collision course with the autonomous vehicle, or it may halt directly in the planned path and not move again. Allowances must be made for these situations and the appropriate strategy must be chosen in order to prevent damage to the vehicle. If the vehicle cannot get around the object, then it may have to abandon its mission goal and attempt to accomplish other goals. Strategies such as these must be made available to the autonomous vehicle so that it can perform its mission effectively. Selecting and executing the appropriate strategy will be a cooperative effort between the

operating system and the mission planning mechanism.

Scope

This thesis effort will concentrate on developing an operating system suitable to demonstrate the mission planning and execution function of an autonomous vehicle. The effort will be limited to identifying only high-level functions the operating system must perform, such as handling accesses to a shared common memory and calling on appropriate specialists.

A mission planner will be developed suitable to demonstrate the interactions between the operating system and a planner mechanism. Actual detailed plans will not be developed in an effort to focus on the concepts involved.

Finally, concepts from a path planning program developed in a previous thesis effort will be incorporated into the overall program in order to demonstrate the operating system and mission planner's capabilities.

Assumptions

This research effort assumes that the autonomous vehicle will have basic sensory capability in its design. Sensory mechanisms necessary to detect objects, orient the vehicle, and compute distance travelled will be simulated in the software algorithm. Furthermore, the autonomous vehicle will be modelled as a multiprocessor system composed of four microprocessors. Research work at AFIT in the area of autonomous vehicle design supports this assumption (3; 12).

Summary of Current Knowledge

Researchers in the field of cognitive psychology have studied how humans construct plans to accomplish errands. Their research revealed certain techniques humans use in constructing plans to accomplish a number of errands during a single day. In an attempt to emulate human behavior in these situations, they developed a computer program using the Lisp programming language. The algorithm they developed could emulate the behavior of several of the test subjects (8:2-3). In their model, the planning process was composed of the independent and asynchronous operation of many distinct specialists or knowledge sources. Each specialist made tentative decisions for incorporation into an overall plan. These decisions were then stored in a shared common memory, or blackboard, for use by other specialists. The blackboard was partitioned into different levels with each level representing conceptually different categories of decisions. Some of the specialists were not restricted to accessing single levels of the blackboard. Instead, they could access other levels and base their decisions on the decisions of other specialists. Hence, information could be shared and decisions altered based on new information gained from interacting with other specialists. This process was termed opportunistic planning and reflects the techniques humans use in constructing plans to accomplish everyday activities.

The idea of cooperating specialists was used in Reference (6) to implement the operating system of an autonomous vehicle. Their operating system consisted of cooperating expert modules together with a high-level coordinator that invoked the

appropriate expert module based on the current state of the robot. Some of the tasks performed by the expert modules included: error processing, mission planning, and path planning.

In the area of planning, several types of planners have been developed for domains ranging from game playing to assembly tasks (4). Of interest to this research effort are the planners that have been developed to carry out everyday human tasks. In particular, Wilensky (15; 16) has studied the relationship between planning and understanding in his research in the area of natural language text understanding. In problem solving, a planner is given a goal and must construct a plan to satisfy it. In contrast, an understander might need to follow the plans of an actor and make inferences about the actor's goals. Similarly, a robot may be given several tasks to perform, and it must construct plans to accomplish these tasks. When to perform these tasks is a function of the robot's ability to make inferences based upon conditions in its environment. Likewise, if more than one robot is present, it may be advantageous to infer what task the other robot is performing so consideration can be given to the changing environment.

Certain portions of the knowledge required to construct plans and to understand plans can be shared between a planner and an understander (15:29-40). This knowledge consists of knowledge about the planning process itself and Wilensky termed it meta-planning knowledge. He organized this knowledge under four main principles he termed meta-themes:

1. Don't waste resources

2. Achieve as many goals as possible
3. Maximize the value of goals achieved
4. Avoid impossible goals

These themes reflect some of the basic underlying principles humans use to guide them in constructing plans. For example, the meta-theme don't waste resources may arise when a person is given many tasks to perform and only a limited amount of fuel in his vehicle. The person would then try to plan an optimum course of action to follow in accomplishing these tasks. If some unforeseen event occurs, such as a traffic jam, the person may realize that he won't be able to accomplish all the assigned tasks. Therefore, he may use the second meta-theme, achieve as many goals as possible, in order to abandon certain tasks that he considers insignificant. These themes serve to guide the selection of goals, and hence, the appropriate plan. Wilensky implemented his concepts in a story understanding program known as PAM. The concepts he developed would be useful in a mission planner in an autonomous vehicle since the ability to infer goals and to re-plan actions if necessary is vital in a dynamic environment such as an aircraft flight line.

Associated with the mission planning function is route planning. A program was developed by Monaghan (10) that determines the optimum path between points in a simulated aircraft hangar environment. The environment was modelled as a two-dimensional representation of real world edges. A complex configuration space was derived from this world model and represented the free space that the autonomous vehicle could move through. Using the

configuration space, point-to-point motion was then planned to the destination. As mentioned earlier, however, all objects in the environment were stationary; therefore, the robot always had an accurate world model representation. Nevertheless, the program provides the necessary function of route planning, and unforeseen situations such as moving objects might best be performed by the operating system itself.

Current work in the field of autonomous vehicle control supports much of the approach taken here. However, differences exist in the philosophies of representing knowledge. Specifically, researchers at Hughes Laboratory have taken the approach of representing the knowledge base in the form of a production system (9). They categorize the knowledge in an autonomous vehicle as being composed of special problem solvers, scripts, and domain-specific production rules. The special problem solvers perform the path planning function mentioned earlier. The scripts, on the other hand, are symbolic representations of stereotypical sequences of events. How to use these scripts is encoded in the production system rules in the form of IF-THEN statements. Therefore, all strategies on how to plan are grouped together in the production system, unlike the approach taken in metaplanning mentioned earlier. The approach they take reduces the level of abstractions in the planning process to basically two levels: find an appropriate production rule, then use the scripts encoded in that rule. The approach taken, however, is a vast improvement over previous problem-solver based systems in that this method is knowledge driven.

Approach

The problem of autonomous vehicle planning and navigation is broken down into three tasks:

1. Operating system development
2. Mission planner development
3. Route planner development

The operating system has the responsibility of controlling communications between the microprocessors in the vehicle and invoking appropriate routines. Therefore, the operating system provides coordination among specialized routines, and as such, will be developed first to provide the driving mechanism for the mission and route planner. Specialized routines in the operating system will be developed to provide necessary functions such as controlling accesses to memory, monitoring the execution of a planned route, monitoring fuel status, and issuing movement commands. Implied in this is the development of a blackboard to provide the means for communication between processes. The partitioning of the blackboard will reflect the categories of decisions made by the specialized routines. For example, an execution monitor specialist might access the planned-route partition, as well as the current-state partition in an effort to determine if the vehicle is on course. The specialists will not be restricted in their accesses to partitions; however, direct communication between specialists will be controlled by an overall scheduler specialist.

Once the operating system has been defined the mission planner will then be developed. A version of Wilensky's PAM

program, known as Micro-PAM, will be expanded and modified to handle the domain of the autonomous vehicle. Micro-PAM will be modified to handle multiple tasks and the possible interactions between these tasks. Furthermore, in an effort to increase its efficiency, an indexing scheme will be added to the program in order to reduce the search time through the rules in its knowledge base.

Finally, a simple route planning program will be developed in order to demonstrate the concept of planning in a dynamic environment. Although the primary emphasis of this research effort is on the interactions between the operating system and mission planner, a mechanism for generating routes through the environment is needed. Therefore, a simple world model will be derived along with pre-planned routes for interfacing with the overall program.

Equipment Requirements

All work will be done on the AFIT VAX 11/780 (SSC) running under the UNIX operating system. The VAX currently supports the ROSS programming language and Franz Lisp, which will be used for the operating system and planning system design respectively. No problems are anticipated in integrating the operating system and planning system since the ROSS language is compatible with Franz Lisp. Nevertheless, interface points will be defined in an effort to keep the two systems as independent as possible.

II. Autonomous Vehicle Operating System

Introduction

In the early days of computers, one might have defined an operating system as the software that controls the hardware. Today, there is a significant trend for functions to migrate from software to firmware, or microcode. Thus, a definition of an operating system today might be the software and firmware that make the hardware usable. The hardware provides the computing power while the operating system makes this power available to the user (1:5).

An operating system is primarily a resource manager, and the resource it manages is the computer hardware. Some of the tasks the operating system performs are:

1. Scheduling processor time
2. Scheduling storage access
3. Recovering from errors
4. Facilitating input/output
5. Interfacing users

Most computer users are familiar with the last function mentioned: interfacing users. The operating system provides a friendly interface with the actual computer hardware, and most users today are unaware of the operating system's other functions. The other functions, however, are some of the operating system's most important functions, and the ones that are of

most concern to this thesis effort.

The other functions represent low-level tasks that the operating system must perform in order to make the hardware usable. Scheduling processor time is an important function in a multi-user computer environment; it allows many users access to the computer's main resource -- the processor. Likewise, scheduling the users access to storage devices is equally important in the multi-user environment because it allows data to be readily available for the processor's use. Finally, recovering from errors is a vital function to the smooth operation of the multi-user computer environment. It frees the human supervisor from the job of restoring the computer to operational status each time a tape drive or disk was unavailable to a user. Thus, it allows the supervisor to concentrate on more serious errors, such as hardware errors.

As mentioned earlier, these tasks are low-level tasks important to the smooth operation of the computer. In an autonomous vehicle, low-level tasks such as these could be performed by a conventional operating system. However, with the addition of intelligence to the vehicle, new functions must be performed that cannot be handled by a conventional operating system. More flexibility is required in the operating system in order to handle situations that may not have been known a priori (11:3). Conditions such as hardware errors and unavailable storage devices can be anticipated and allowances can be made. However, when specialized functions are being performed in an autonomous vehicle, such as route planning, and a previously unanticipated

condition in the environment arises, what actions should be taken? Should available fuel resources be shared equally among all on-board devices if the situation is threatening to the vehicle? Or, should maximum effort be expended in planning evasive maneuvers? If an evasive maneuver strategy is decided upon, attention should be given to the mission planning function, and unnecessary on-board equipment should be shut down. Deciding what course of action to follow would be the job of an intelligent operating system. The more conventional operating system could easily handle the error conditions that might arise when it tries to access equipment that has been shut down. So, what is needed is a more flexible operating system that can deal with the specialized intelligence functions in an autonomous vehicle.

The specialized intelligence functions in an autonomous vehicle can be broken down into four categories:

1. Mission planning
2. Route planning
3. Execution monitoring
4. Error handling

Each of these categories may have associated with them subspecialties that are vital to the correct operation of other specialized functions. Nevertheless, there is a need to drive each of these mechanisms in the proper sequence and not just in a straight-line fashion. Hence, the need for an intelligent operating system to control these functions.

The last function performed by a conventional operating system mentioned earlier is communicating with users. An autonomous vehicle may need to communicate with a human in order to

receive new instructions; however, the process is a specialized function based on the available I/O devices and language support. Communication with an autonomous vehicle might be through a cable data-link between a main-frame computer and the vehicle, or through an on-board input device such as a keypad (12:8-9). Various other I/O devices have been explored and future progress in speech recognition may allow the human supervisor to talk directly to the autonomous vehicle.

The language used for communication might be a specialized control language (11:10), or the system might support a natural language interface. In any case, the mode of communication is a function of the design of the autonomous vehicle, and is a specialized function of the operating system that will not be considered here. However, the design of the autonomous vehicle is also important to the other tasks that the operating system performs. Therefore, vehicle design should be considered in any operating system implementation. The effort here was to keep the operating system as general as possible; nevertheless, a model of an autonomous vehicle was used as a basis for many of the functions in this operating system.

Autonomous Vehicle Model

To serve as a basis for the operating system functions, the AFIT Mobile Autonomous Research Robot System (MARRS) was used as a model. The AFIT robot has been the subject of several thesis efforts at AFIT and has undergone numerous changes from its basic configuration as a Heathkit HERO-1 robot (3). Many of the changes have been enhancements of the basic system in order to

accommodate additional hardware. The available memory has been expanded in order to provide additional storage space for the software required to drive the new hardware. Furthermore, the original carriage has been enlarged since the hardware enhancements used up the available space in the basic carriage.

Many types of sensor devices have been proposed and implemented on robots in research institutions and industry. The sensors may perform their functions through mechanical, optical, acoustic, electric, or magnetic means (11:6-9). Numerous types of sensor devices are available, and each one provides certain advantages and disadvantages. Regardless of the type of sensor, effective use of the sensors is required of an intelligent vehicle in order to accomplish its mission goals. The sensors, however, should provide the capability to detect obstacles, orient the vehicle, measure the distance travelled by the vehicle, and determine when objects have been grasped.

The AFIT MARRS robot has some of these capabilities. Specifically, the MARRS robot has been modified to include the following sensors:

1. Thirty-two Polaroid sonar transducers
2. Optical shaft encoder subsystem
3. Laser barcode reader
4. Gyro-compass

These sensors allow the robot to detect objects in the environment, locate its position in the environment, and determine its heading. Therefore, the sonars, optical shaft encoder, and the gyro-compass were selected for modelling in this thesis effort

because they represent the minimum sensors required in order to navigate. In addition, an end-effector sensor was selected for modelling in order to give the vehicle the capability to determine when objects have been grasped.

The MARRS robot is a multiprocessor system based on the Motorola 6800 family of microprocessors (3:8). A separate microprocessor controls the various sensors while another microprocessor acts as the main drive computer. Because the MARRS robot uses the 6800 family of microprocessors, its memory addressing capability is very limited. The basic software required to support the sensors and the drive mechanisms leave limited storage space for any elaborate planning and navigation software. Therefore, a method is needed to conserve memory space and facilitate processor to processor communication. The approach taken in this thesis effort was to use a shared common memory, or blackboard, in order to allow processor to processor communication and minimize duplication of data throughout the system.

Intelligent Operating System Tasks

Hayes-Roth (8) has formulated a planning model in an attempt to emulate human behavior in constructing plans to accomplish errands. An important concept that arose during their research was that of opportunism. Human subjects were given several errands to perform and they were to construct a plan to accomplish all the errands in a specific amount of time. The errands consisted of routine activities people perform everyday such as going to a grocery store or a bank. The subjects were given a

map of a city with the locations of each errand, and the methods they used to construct their plans was recorded.

The methods the subjects used to begin their task varied. However, throughout the planning process the subjects used similar methods to arrive at a solution. A subject may have begun the planning process by making abstract decisions about features of the plan such as accomplishing all errands in the northwest part of the city first. While carrying out the errands in that part of the city, he may notice an opportunity to easily accomplish another errand that may not have been in the northwest part of the city. He may then decide to construct a plan around accomplishing errands that are close to each other, so he may formulate a plan around clusters of errands. This ability of humans to switch from an abstract level of decision making to a detailed level while constructing plans was the driving force in the formulation of Hayes-Roth's planning model.

The model they proposed was based on certain features of the Hearsay-II system (8:376) such as:

1. Multiple cooperating knowledge sources
2. Opportunistic problem solving behavior
3. Communication via a blackboard
4. Scheduler to control activities

In their model, the planning process was carried out by many different specialists, or knowledge sources, each making tentative decisions for incorporation into an overall tentative plan. The planning activity was controlled by an executive routine that scheduled specialists and invoked the appropriate ones based on

certain policies governing the overall plan. For example, if the policy is to construct the most efficient plan, then the executive might schedule the specialist whose job it is to find errands that are close together. The executive is driven by policies that the plan must meet, and the planning process is only stopped once the executive determines that they have been met.

An executive controller, a blackboard, and planning specialists are the concepts that are the main influence to this thesis effort. The blackboard approach requires the operating system to arbitrate memory accesses. When is the data in the blackboard accurate? If the vehicle is in motion, sensor data in the blackboard may be changing rapidly and might be of limited use to a mission planner if the sensors were denied access to the blackboard. What are the points at which other functions should be denied access to the blackboard, and the data in the blackboard is considered accurate for a particular function? Arbitrating blackboard accesses based on prevailing conditions is essential to effective operation of the autonomous vehicle.

Just as the operating system must decide what function should have access to the blackboard, it must also decide when to schedule functions. During startup, the mission planner would be invoked in order to plan the sequence of actions to accomplish its mission goals. When actions have been planned, the route planner should be invoked next in order to find an optimum path to the destination if required. When motion begins, the execution monitor should be invoked in order to ensure the vehicle is

on course. If the vehicle strays from its assigned course, an error handling function would be invoked in order to correct the deviation. If a more severe error is detected, such as an imminent collision, the mission planner would be invoked in order to select an appropriate avoidance strategy. Scheduling and invoking the appropriate functions is therefore a required task of the operating system and must be based on prevailing conditions in the environment as well.

There are other tasks that the operating system should perform, such as communicating with the vehicle's conventional operating system; however, the two tasks discussed previously have been chosen as basic tasks required to demonstrate the interactions between the specialized intelligence functions in an autonomous vehicle. As such, this thesis effort will concentrate on these two tasks in order to allow the development of a mission planning system.

III. Planning

Introduction

Planning is the process of using a problem solving procedure to determine a course of action prior to executing those actions. Failing to plan can lead to less than optimal problem solving, and in some cases, can preclude ever finding a solution to a problem. This is especially true when certain aspects of a problem may interact with each other implying there is a required sequence in which actions are to be performed. If an autonomous vehicle had the tasks of refueling an aircraft and repairing a hole in the aircraft's fuel tank, the order in which these tasks should be performed is clearly important. Repairing the hole in the fuel tank by using a welding torch could have serious consequences if the autonomous vehicle had just finished refueling the tank. Each problem may have been solved, but a new problem has been created from the resulting explosion of the aircraft.

Planning, then, involves knowing how the planner's universe may change as a result of some action, and how these actions may interact with other aspects of the plan (13:249). So, other than needing this form of knowledge, the planner must use a problem solving procedure that can analyze aspects of a plan, and thus, guide him in selecting the appropriate actions. In complicated problem domains, this is essential since there may be

numerous interactions between goals in the plan, and disastrous consequences could result from an improper selection of actions.

A method that is used in planning is to decompose the problem into smaller subproblems (13:248). Each smaller subproblem can then be worked on, and the solutions can be combined to form an overall solution. Decomposing a problem makes it more tractable, and it is a method that is widely used in engineering disciplines. Two approaches to planning that make use of decomposition in varying degrees will be discussed in order to provide a foundation for the planner designed in this thesis effort.

Next, an approach used in the problem domain of story understanding will be discussed since some of the concepts used in that approach have been incorporated in this thesis effort. Finally, a brief discussion of system organization will provide a foundation for the organization adopted here. While much of the system organization discussion relates to the operating system design, the operating system itself can be considered a planner, and as such, the Hayes-Roth (8) model will be examined once again.

Hierarchical Planning

A hierarchical planner decomposes a problem into a hierarchy of representations. These representations are an abstraction of the problem and its subproblems. An example of such a hierarchical representation is shown in Figure 1 for the problem of planning a days activities.

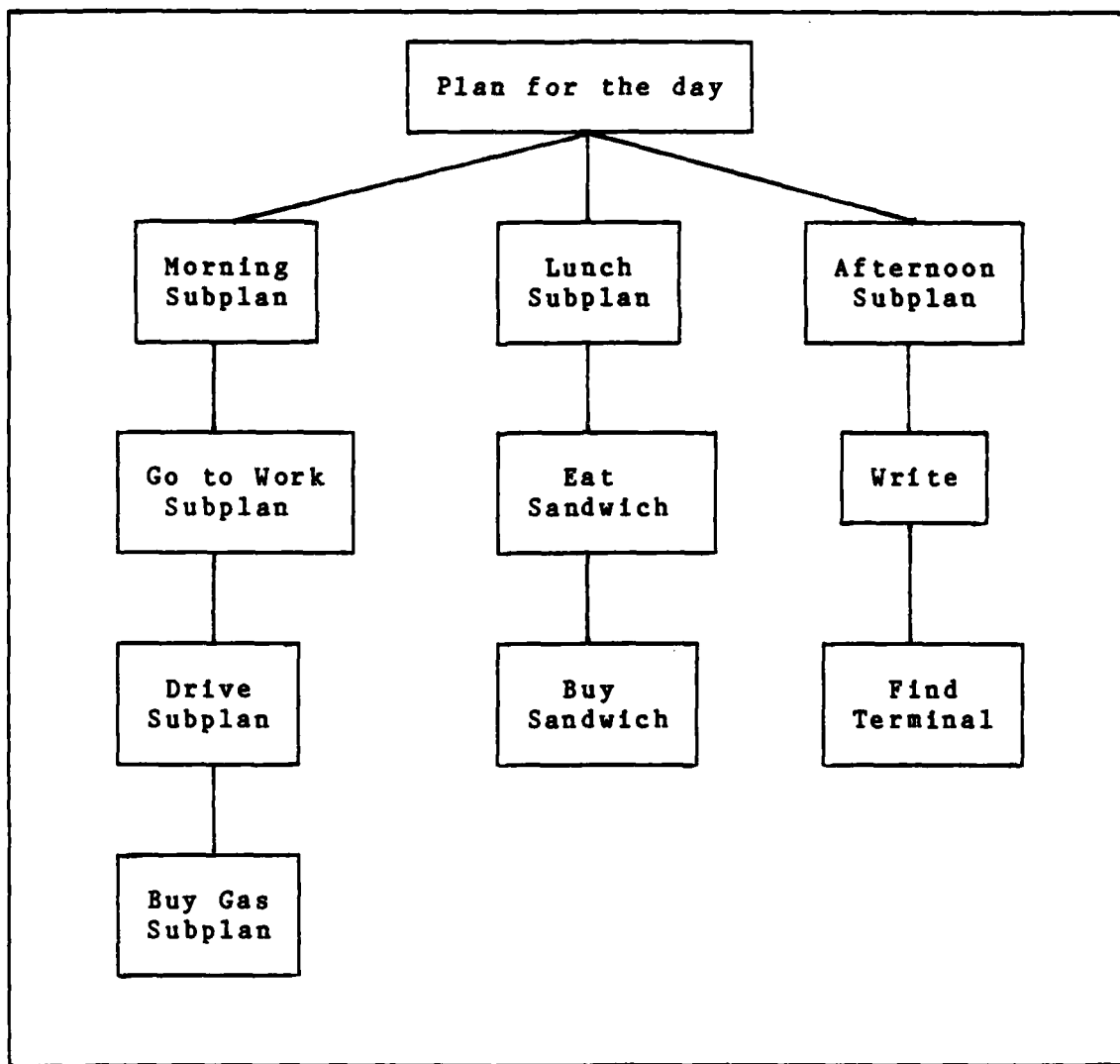


Figure 1. Plan for Days Activities (4:516).

The plan in Figure 1 is shown as a hierarchical structure of subplans with the specific details on how to accomplish the subplans left out. Instead, the subplans represent an abstraction of the activity to be performed with each lower level being abstractions of the details to accomplish the higher levels. For example, under the morning subplan, the plan for going

to work involves buying gas and driving. But, buying gas requires having money; therefore, a subplan for getting cash is required immediately under the buy gas subplan. Getting cash, however, is an abstraction of the solution, so the next level under that might be a go to the bank subplan. In each case specific details have been left out. Going to the bank involves walking to the car, getting into the car, driving to the bank, and so on.

The advantage of leaving the details out initially is that critical subgoals of a problem can be considered first. By considering the critical subgoals first, the details can be added in later as the general plan is formulated. This helps in reducing the required search of the problem space by developing the plan at a level at which it is not computationally overwhelming (4:517).

For a very complicated problem, the planner could easily get bogged down if it tries to consider all possible goals in the problem space at once. Trying to formulate a plan by first considering a goal of opening the car door may or may not lead to an acceptable overall plan. Instead, dealing with abstractions of the problem helps to guide the search and reduce the inefficiencies that would result from starting at a high level of detail.

Besides trying to limit the search of the problem space, there is the problem of interacting goals. The order in which actions are to be performed is sometimes important, as in the case of the autonomous vehicle with the tasks of repairing and refueling the aircraft's fuel tank. Two techniques have been

used to deal with the problem of interacting goals.

The first technique relies on the assumption that goals are independent and can be achieved sequentially in any order. The planner arbitrarily orders the goals and then goes back and tries to repair the plan if goal conflicts are discovered. This method is useful if there is no a priori knowledge about the proper ordering of goals, and it helps reduce the combinatorial explosion of trying to initially determine the proper order of numerous goals (4:520).

The planner using this technique for the example of the autonomous vehicle tasks of repairing and refueling an aircraft's fuel tank might start the planning process by using an arbitrary ordering of the tasks. If it starts out by planning to refuel the aircraft first, later on while developing the plan to repair the fuel tank, it may discover that welding the fuel tank conflicts with a full fuel tank. To repair the plan, the planner would reorder the goals by doing the repair fuel tank plan prior to refueling.

In the previous example, complete plans were developed separately for the two tasks; however, in some cases it may be better to intertwine plans. This type of planning is known as non-linear planning, and it is the second method for dealing with interacting goals (13:267). A non-linear planner would not arbitrarily order goals. Instead, plans might be developed in parallel, and the interactions of goals analyzed as the planning proceeds. Figure 2 shows what a non-linear planner might do for the two tasks of repair and refuel aircraft's fuel

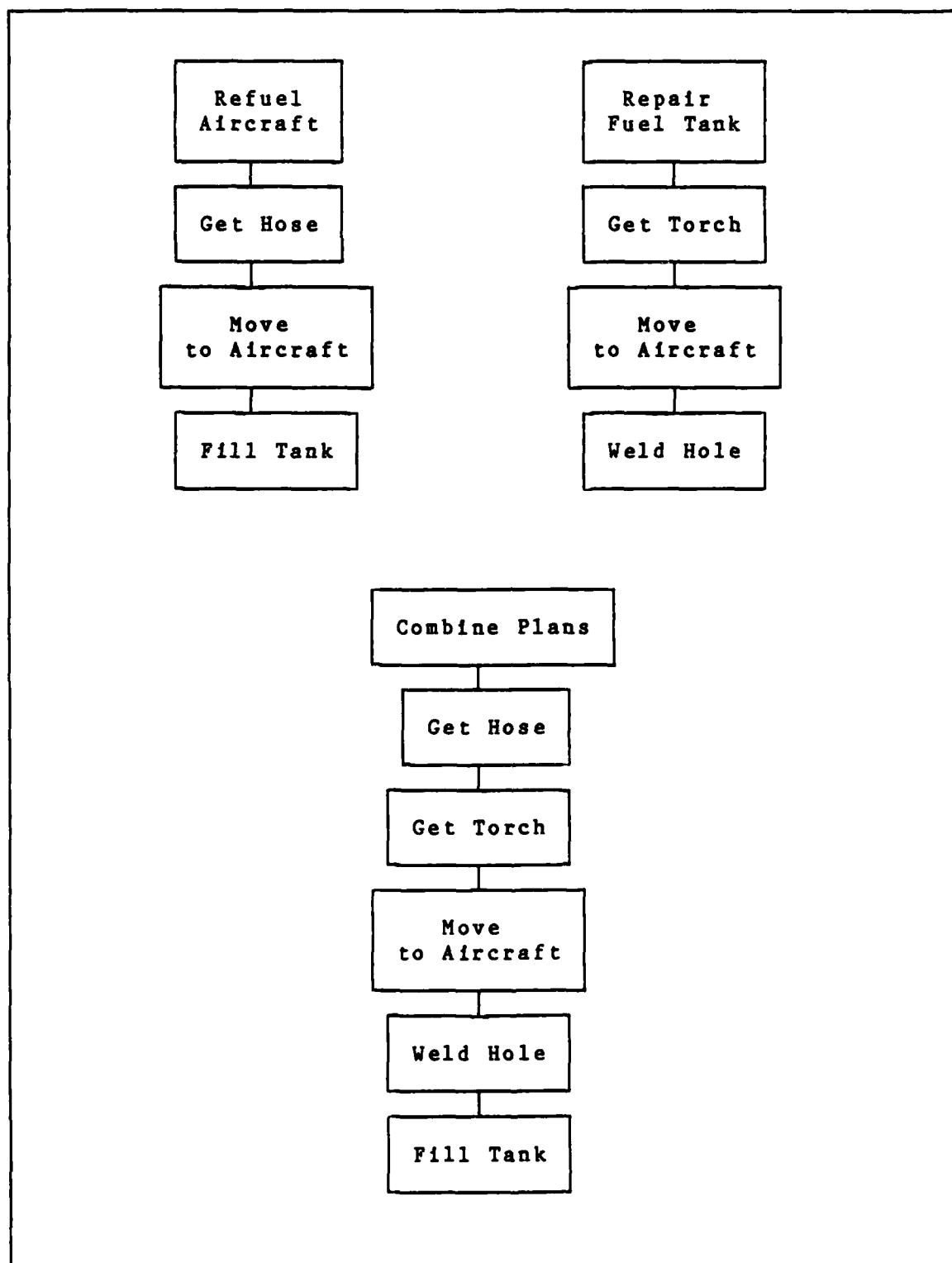


Figure 2. Non-linear Planning.

tank.

Initially, no order of goals is assumed so the planning proceeds in parallel, and at each step a set of critics examine the plan for interacting goals. The critic notices that a move to the aircraft is called for in both plans, so this step is combined so that only one move to the aircraft is performed. The vehicle would get the hose first, then get the torch, and finally move to the aircraft. At the last step of the plan, the critic notices that welding the goal conflicts with a full fuel tank, so it proposes to do the welding prior to filling the fuel tank. Now this may not be a safe procedure, but it illustrates the concept of non-linear planning. By using a set of critics to examine goal preconditions, intertwining plans can be developed for the conjunctive tasks of refuel and repair fuel tank.

Figure 2 also illustrates the second method of plan construction: nonhierarchical planning. The problem of interacting goals is common to both types of planners; hence, the discussion here also applies to nonhierarchical planning.

Nonhierarchical Planning

A second method for constructing plans uses a nonhierarchical approach. The term nonhierarchical is misleading in that it implies no decomposition is used in the plan construction. This is not the case; both hierarchical and nonhierarchical planning use a hierarchy of representations in plan construction. The difference is in the level of representation used in the decomposition. A hierarchical planner generates a plan structure in which the highest levels are very

sketchy, and the lower levels are very detailed. A nonhierarchical planner, on the other hand, does not distinguish between aspects that are critical to the success of a plan, and those that are only details (4:517). Figure 2 illustrates a nonhierarchical plan and the drawbacks associated with such planning.

Each step in the plan shown in Figure 2 is really a detail that could be filled in later once policies governing the plan have been handled. For example, it might have been better to start the planning process at a higher level of abstraction, such as considering the safety aspects first. The planner would have established a subgoal of being safe, and lower level subgoals would have arisen hinged on this policy of being safe. Potential lower level subgoals might have been related to eliminating dangerous conditions such as fuel vapors and sparks. Proceeding from this level of abstraction would have guided the search for a solution and helped to reduce the possibility of having to reorder goals. Goal interactions may still occur; however, the critical aspects of the plan are considered first.

Nonhierarchical planning, therefore, is not of much benefit in complicated problem domains where numerous goal interactions may occur. The planner gets bogged down in sometimes futile searches of the problem space in an effort to consider all goals at only one level of abstraction, rather than a hierarchy of abstractions (13:271). Nonhierarchical planning, however, was used in some of the earliest planners, and it is mentioned here to serve as a contrast to hierarchical planning.

Metaplanning

Planning, therefore, is the process of deciding on a course of action. There is a richer knowledge, however, that human planners use in this process that has not been discussed. What constitutes a good plan, and what are some of the techniques human planners use in constructing good plans? This type of knowledge is knowledge about the planning process itself, and Wilensky (15; 16) used this knowledge, known as meta-knowledge, in his study of natural language text understanding.

Wilensky used as a basic premise for his research the fact that in order to understand stories, the understander must have knowledge about the planning process. As an example, consider the following story:

John was driving to the store when he noticed a tornado coming his way. He immediately turned the car around and drove back home.

A possible explanation for John's actions might be that he considers preserving his life more important than what is ever at the store. But, what is needed to arrive at this explanation? The answer is in how humans formulate plans.

From the first statement, we can infer that John's goal was to be at the store, and his plan to achieve that goal was to drive to the store. From the second statement, we can infer that John's goal was to be at home, and his plan was to drive home. This, however, does not explain John's actions. We also need to know that tornadoes can be life threatening, and humans have a desire to avoid such situations. With this knowledge, added to

the knowledge that humans use to construct plans, we can arrive at the above mentioned explanation.

Figure 3 illustrates the planning process John may have used in deciding what to do in his situation. He had two goals, to preserve his life, and to be at the store. He realized that these two goals were in conflict with each other, so he used his knowledge about planning to resolve this conflict. An acceptable

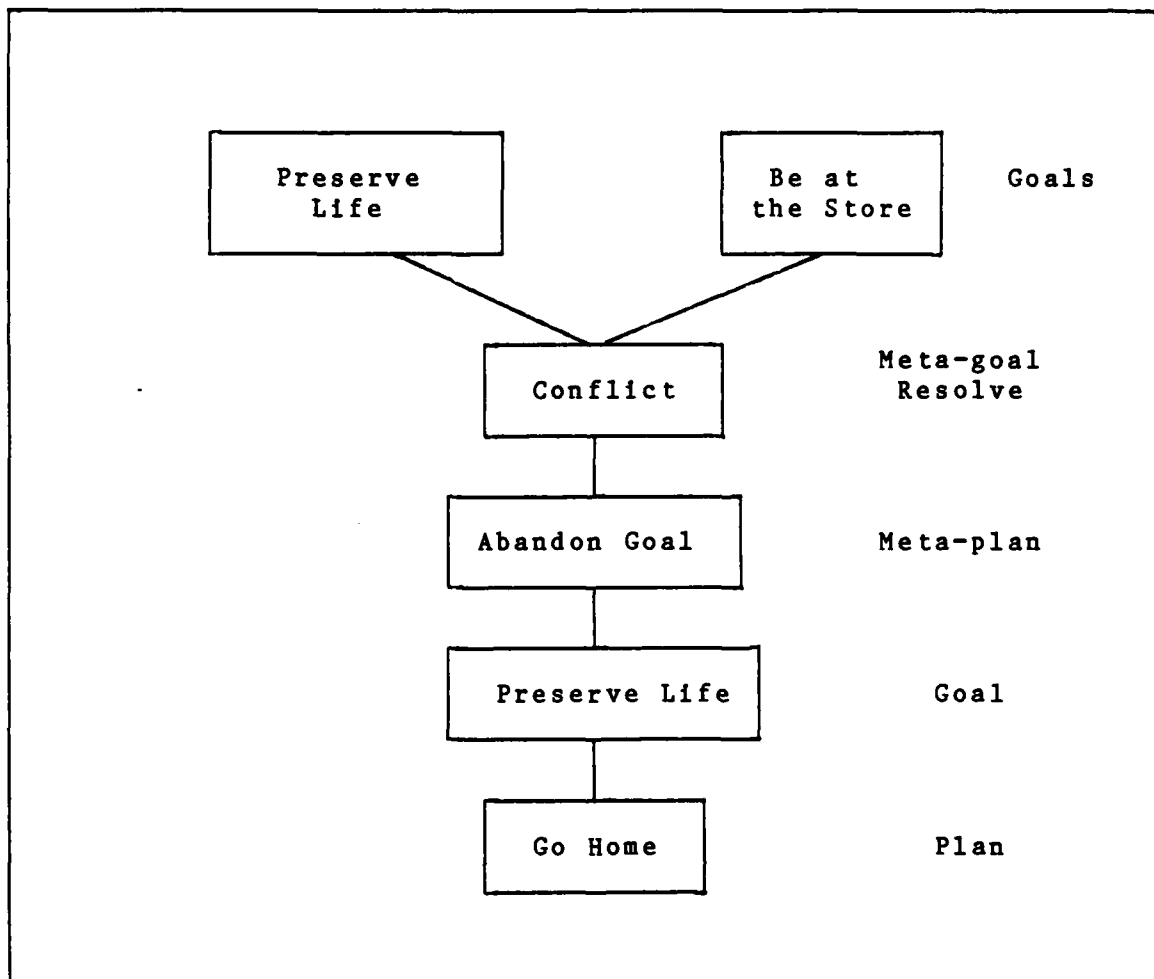


Figure 3. Meta-planning Process.

procedure to use in situations where conflicting goals occur is to abandon the goal that is considered least important. John used this planning knowledge to abandon the goal of being at the store. With only one goal left, he constructed a plan of driving home to achieve that goal.

Now, with this added knowledge about how humans plan, we can arrive at the explanation mentioned previously. We needed to know that humans sometimes abandon goals in favor of more important ones. This knowledge is used in many types of problem domains, and it is not just applicable to the above story. Likewise, a good planner must have this knowledge available in order to construct acceptable plans. The knowledge, therefore, can be used by both an understander and a planner if it is in a form that can be shared (16:31).

Wilensky organized this knowledge around four themes he called meta-themes (16:31). Each meta-theme gives rise to goals, known as meta-goals, for which meta-plans might be applicable. The four themes are:

1. Don't waste resources
2. Achieve as many goals as possible
3. Maximize the value of goals achieved
4. Avoid impossible goals

John, in the previous example, realized that he had conflicting goals, so the meta-theme maximize the value of goals achieved came into play. The meta-theme initiated the meta-goal of choosing the most valuable scenario. A possible meta-plan to

achieve this meta-goal is to simulate the outcome of each conflicting goal and then select the most valuable outcome (15: 72; 16: 209-211). John simulated the outcome of preserving his life and decided this was the most valuable scenario as compared to being at the store.

Meta-themes, which give rise to meta-goals and meta-plans, can be used by both an understander and a planner. Since a truly intelligent autonomous vehicle will need to understand, as well as plan in any complex environment, it is these concepts that are most appropriate here. However, in order to use these concepts, the planner must be able to detect that it has goals, as well as generate plans to accomplish them. Wilensky proposed a planner based on the following elements:

1. Goal detector
2. Plan generator
3. Executor

The goal detector is the inferencing mechanism that passes the goals to the plan generator. The executor simply carries out the plans proposed and detects any errors in the plans (16: 215).

We now have the elements an autonomous vehicle can use to operate in a complex environment. The inferencing mechanism can use knowledge about the planning process to understand situations that may occur in the vehicle's assigned tasks, especially if it is working in conjunction with other autonomous vehicles. But, by providing it with the capability to understand its environment, the planning process can be guided more intelligently. What is needed now is a way to organize and direct these various

elements of the planner.

System Organization

One of the simplest ways to organize a planning system might be to write rules containing all the knowledge, and then let the planner use those rules to find a solution. In a complex domain, however, the number of rules can be large, and the system might become bogged down in searching through the knowledge base. Therefore, an organization scheme is needed that splits up the knowledge in a large domain into separate modules. The blackboard approach is one such scheme, and it was discussed earlier in Chapter II in the context of an operating system. It is now looked at in a little more detail in the context of organizing and controlling the activities in the planning system.

As mentioned earlier, the blackboard provides a means of communication between knowledge specialists in a planning system. The elements of a planner mentioned in the last section, a goal detector, plan generator, and executor, will all need to be controlled, and the knowledge they use can be shared. The goal detector can use knowledge about plans to infer goals, while the plan generator can use the knowledge to construct the plans. Likewise, the executor can use the knowledge to simulate what might happen carrying out the plans in its effort to detect errors. So, the blackboard can be used for communication, but the three elements must still be invoked at the proper time. Hence, the need once again for the operating system to direct activities.

Hayes-Roth's planning model used an executive controller to

direct activities in the planner (8:380). The knowledge specialists in the Hayes-Roth model had associated with them a set of triggers that specified conditions under which they should be activated. When conditions were favorable for a knowledge specialist, they were added to an agenda list of knowledge specialists to be invoked. Based on policies about the overall plan, knowledge sources were scheduled depending on which knowledge specialist had the ability to deal with current policies. Once a schedule was decided upon, the knowledge sources on the agenda were invoked, or fired, and they used knowledge in the blackboard to formulate hypotheses. By formulating hypotheses and writing them to the blackboard, the knowledge specialists might alter conditions; thus, other knowledge specialists might trigger and be added to the agenda list (8:380; 13:281).

In this manner, activities in the planning mechanism are controlled based on prevailing policies. The planner design discussed previously can likewise be directed in its planning process. The goal detector might work in conjunction with an execution monitor to interpret conditions in the vehicle's environment. Goals that the detector generates might trigger the planning generator and cause it to be added to an agenda. The goals may not always trigger the plan generator, but they could trigger another knowledge specialist such as an error handler. Likewise, the plan generator might trigger an executor as a result of the type of plan generated. If the plan involves a certain amount of risk, it might be wise for an executor to simulate the plan and detect unwanted occurrences. Nevertheless,

the activities are directed by the operating system based upon the vehicle's mission and current conditions in the environment.

IV. Operating System Design

Introduction

The factor influencing the design criteria for the intelligent operating system was the need to control blackboard accesses and schedule appropriate knowledge sources. Underlying these two functions is the relationship between a mission planning mechanism and the operating system. The two are not totally independent. As mentioned in the discussion on planning, there is a need for a goal detector and executor, as well as a plan generator. Just as the duties of planning and understanding can be shared by one mechanism, so can operating system, goal detection, and executor tasks be shared by one mechanism.

A natural level at which to detect goals is at the operating system level. Since the operating system is essentially an interface between the real world and the computing environment, the interpretation of sensory information at this level can be readily transformed into appropriate goals for a plan generator. Likewise, since it is the responsibility of the operating system to carry out plans generated by the planner, it is appropriate to place the tasks of executor at the operating system level to detect any errors in the proposed plan. Once again a relationship exists between two tasks: the executor function requires the ability to detect goals. Therefore, added support is given to the decision to group these functions at the

operating system level.

There are further arguments for placing these two tasks at the operating system level other than their inherent relationship. As discussed in Chapter III, the function of the executor is to carry out the plans and detect any errors in the proposed plans. This is the concept of plan projection that Wilensky proposed for his planning mechanism (15:17; 16:218). Since it is the primary task of the operating system to actually carry out the proposed plans, the function of plan projection can be performed at this level through simulation. Prior to running such a simulation, the current state of all resources would need to be saved so that the simulation would not affect vital information. By using the operating system for plan projection, the complexity of the planning mechanism is reduced by eliminating the requirement to duplicate operating system functions at the planning level.

Hence, with these additional tasks the design criteria is formulated based upon the relationship between the operating system and planning mechanism. The tasks to be performed by the operating system are:

1. Controlling accesses to the blackboard
2. Scheduling knowledge sources
3. Performing goal detection
4. Performing plan projection

The task of goal detection was listed separately rather than grouped with plan projection since this task must be performed when the vehicle is actually carrying out plans as well as when

it is simulating plans. Thus, it is not a duty that will be performed solely during plan projection.

The autonomous vehicle model discussed in Chapter II requires the ability to simulate a multiprocessor environment with numerous knowledge specialists communicating via a blackboard. The asynchronous nature of this computing environment influenced the selection of the programming language for the operating system. The ability to simulate knowledge specialists, as well as time progression, were criteria for language selection. The time constraints on this thesis effort would have precluded any development of a programming environment for the operating system in addition to implementing a planning mechanism. However, a suitable language exists that meets the requirements of this thesis.

The ROSS Language

Object oriented programming languages enforce a message passing style of programming. A program written in this type of language is characterized by a set of objects, or actors, that interact with one another by passing messages. The actors have associated with them a set of attributes and message templates that invoke a behavior whenever a message is received that matches a template. The behaviors may be an explicit computation or they may involve passing messages to other actors. In any case, this style of computation is suited to simulations in domains that involve autonomous interacting components.

The ROSS programming language is an object oriented language

whereby processing is done through means of message passing among objects or actors. The ROSS language is ideal for simulating processes in dynamic environments where situations occur in an asynchronous fashion; furthermore, it provides a facility to simulate time, and hence, a means to simulate parallel processes.

In this project, the parallel processes are described as a collection of actors in the ROSS language. Each actor has associated with it a certain behavior, so that when a particular message is received, it will perform some function or ask someone else to perform a function. The actors can be considered autonomous processes, and therefore can be used to simulate individual microprocessors in a multiprocessor computing environment.

Actors are created in ROSS by an explicit call to a built-in ROSS actor. So, if we had a need for a robot actor, then the following ROSS command would result in a robot actor being created:

```
(ask something create generic robot)
```

The actor "something" is the built-in ROSS actor that is present in the ROSS environment at invocation. The preceeding command has now created a robot actor that can be programmed to exhibit certain behaviors when receiving messages. An example behavior for the robot actor might be:

```
(ask robot when receiving (jump up and down)
  (tell user I can't do that))
```

So, when the robot actor receives a message telling him to jump up and down, his reaction is to send a message to the user actor

telling him he can't do that. Likewise, the user actor might have a message template for this situation that causes him to exhibit a certain type of behavior when he is told someone can't carry out a command.

The ROSS language also provides a facility to allow actor's behaviors to be planned. A planned action can be triggered corresponding to a certain time step in the simulation. Thus, events can be triggered that are not the result of receiving a message from another actor. This capability is a convenient way to simulate the actions of a microprocessor that is interfaced to a real world dynamic environment. The computations it performs are due to external inputs and are independent of events in the overall computing environment. Indeed, this facility is desirable during plan projection in order to provide a more thorough analysis of proposed plans. This facility can be used to simulate unexpected occurrences during plan execution in order to investigate the existence of alternative plan scenarios. Planned actions such as a course deviation during motion simulation may have the effect of using up surplus fuel on the autonomous vehicle, and thus necessitate a refueling stop prior to carrying out any of the assigned tasks. By simulating occurrences such as these, a more efficient plan could be developed taking into account the likelihood of unplanned events occurring. Thus, if course deviations will adversely affect the vehicle's fuel status, a plan could be generated that would involve refueling the vehicle prior to beginning other tasks. In this manner, the need to suspend a current task during plan execution is eliminated during plan projection by reordering the

tasks.

These characteristics of the ROSS language were used in both the design of the operating system and the planning mechanism. The planning mechanism itself was considered a separate actor that was invoked and guided by the operating system. However, a different approach was taken in the design of the planning system's architecture in that the internal controlling mechanisms did not rely on message passing. The interface between the operating system and planning mechanism, however, does rely on message passing. Hence, a discussion of the operating system program structure is vital to understanding the planner.

The Program Structure

Figure 4 shows a block diagram of the program structure. The autonomous vehicle is modelled as a multiprocessor system using a shared common memory, or blackboard, for communication between processors. Four processors are modelled: a main processor, and three sensor processors.

The main processor controls accesses to memory and invokes routines to monitor route execution, plan mission tasks, plan routes, etc. Three sensor processors provide heading information, sonar information, and distance travelled information.

The Main Processor. As mentioned earlier, the main processor controls accesses to memory and is responsible for invoking appropriate routines. The main processor, shown in Figure 5, is modelled in the ROSS language as the following

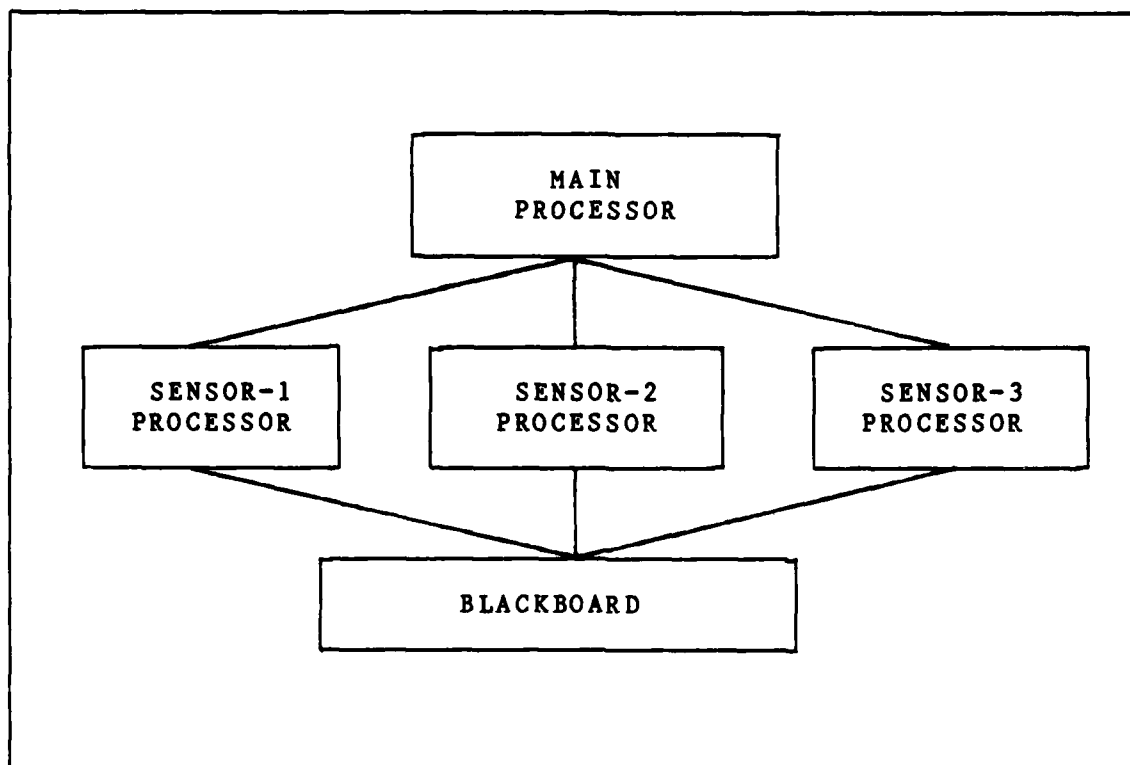


Figure 4. Program Architecture.

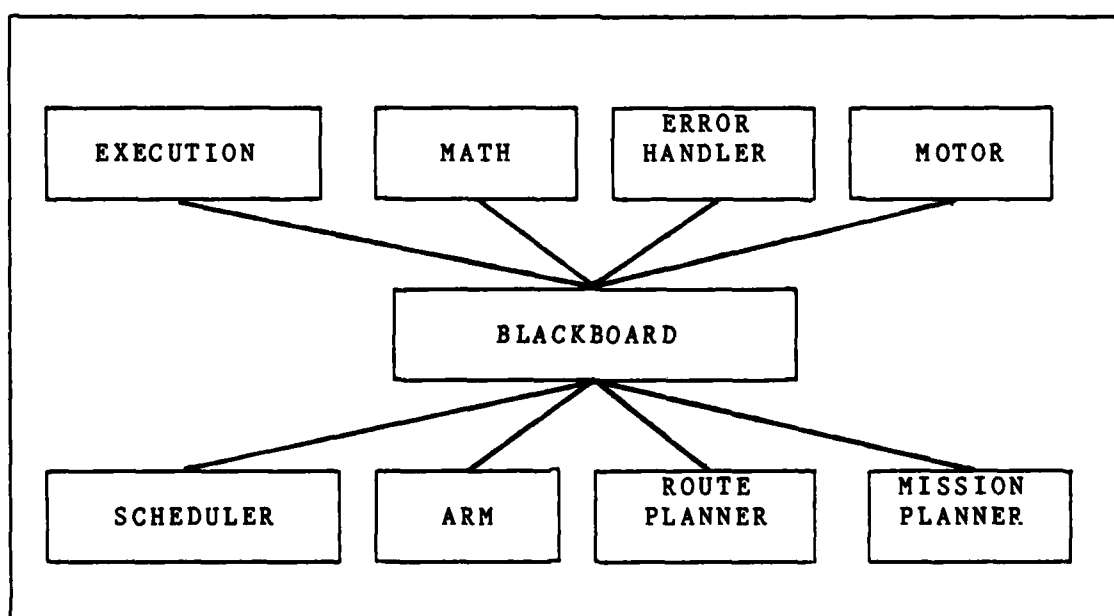


Figure 5. Communication Among Main Processor Actors.

collection of actors:

1. Scheduler
2. Execution
3. Math
4. Errorhandler
5. Motor
6. Arm
7. Route-planner
8. Mission-planner

The scheduler actor is the main actor and is responsible for controlling accesses to memory and updating the clock during the simulation. The scheduler acts as the top level in the overall operating system whose job it is to handle requests for bus access and to ensure that appropriate routines are invoked. It ensures that equal access to the bus is given to each processor during a simulation.

At simulation startup, the scheduler directs the mission planner to formulate a plan based on a certain policy. Once a plan has been created, the scheduler then begins passing messages to the appropriate actors in order to simulate carrying out the plan. Messages are passed based upon an agenda created by the mission planner. The agenda is a list of messages to the actors most capable of carrying out the task. In this manner, the control flow has been initially specified by the mission planner; however, the scheduler has ultimate authority in deciding if

additional actors are required to perform a task.

If a particular actor is unable to carry out a task, then the scheduler detects the type of error involved and modifies, if necessary, the current planning policy and directs the planner to replan a task. Thus, the job of goal detection is performed by the scheduler based upon messages passed from other actors.

When the scheduler detects a task failure, the current agenda is suspended while a new plan is formulated to handle the error condition. Therefore, it is not necessary to start the simulation all over again, but instead action can resume at the point that task failure occurred. Furthermore, any unusual operations required, such as saving the current state data, are suggested by the mission planner during plan generation based upon the type of task failure. Thus, a degree of flexibility is incorporated in the operating system by sharing the duties of program control flow with the mission planner.

The execution actor is responsible for monitoring the motion of the autonomous vehicle. Prior to a motion simulation, the execution actor is called by the scheduler to determine if the autonomous vehicle is at the proper coordinates and the proper heading. The execution actor then notifies the scheduler if the autonomous vehicle is not in the proper location or heading. Otherwise, the execution actor will report that no error exists, and the scheduler will then start the motion simulation.

During the motion simulation, the execution actor is invoked by the scheduler each time the blackboard is updated. The execution actor checks to see if the autonomous vehicle is on course

by comparing the current heading of the vehicle with the planned heading. If any course deviations are detected, they are reported immediately to the scheduler. Finally, the execution actor updates the current coordinates of the vehicle by using the math actor to perform the necessary computations.

The math actor uses information in the blackboard to determine the current location of the vehicle. It takes the coordinates of the start of the current route segment, along with the distance travelled and current heading, and computes the new location of the vehicle. This information is then stored in the blackboard and control is passed back to the execution actor for further processing. Although this computation could have easily been done by the execution actor, future enhancements of the system are expected to require more extensive computations; therefore, this computation was given to a separate actor in order to make upgrade easier.

If a course deviation is reported by the execution actor, the scheduler takes immediate action by invoking the errorhandler actor. The errorhandler's job is to compute a new course heading based upon the planned heading stored in the blackboard. The errorhandler computes the proper heading and then replaces the old value in the planned route segments with the new value. The errorhandler then determines the amount of turn required to reorient the vehicle and then passes this information back to the scheduler for processing.

A separate actor called the motor was created to simulate actual movement of the vehicle. The motor actor processes two types of commands from the scheduler: a turn command, and a move

forward command. If the motor actor receives a turn command, it tells the sensor actor to increment or decrement its heading the appropriate amount. Likewise, if it receives a move forward command it tells the sensor actor to increment its distance travelled attribute the appropriate amount. In this way, changing sensor readings due to movement of the vehicle are simulated. Furthermore, the motor actor decrements the amount of available fuel on the vehicle in order to simulate energy expenditure with each movement command. For the purposes of this project, however, the motor actor is not considered a part of the main processor, but instead represents the actual drive mechanism on the autonomous vehicle.

The arm actor represents the autonomous vehicle's end effector. Its primary function is to update the blackboard on the current state of the end effector. For example, if the actor receives a command to extend the arm to a certain x and y coordinate, the arm actor first checks the blackboard to determine if the arm is already extended, and if so, where. If the actor is capable of performing the task, it updates the blackboard based upon the command it received. Therefore, impossible goals can be detected by the arm actor by comparing the received commands with the current state stored in the blackboard. Hence, error conditions, such as trying to grasp an object when an object is already being held, can be detected.

The route-planner actor is an attempt to emulate the actions of a route planning mechanism. It in no way approaches the complexity of a required route planner, but instead was created

in an effort to provide basic route information. The actor merely retrieves a stored route based upon a message detailing the start of the route and the goal. It has none of the flexibility exhibited in Monaghan's (10) implementation in that it cannot generate routes from arbitrary starting locations. All routes are preplanned and stored by the user in a primitive world model; the route actor merely retrieves the appropriate route. This type of implementation does put constraints on the overall planning mechanism in that it limits the type of movement allowed. However, the implementation was sufficient to demonstrate the other key mechanisms, and as such, provides a high level look at the type of interfacing between the operating system, mission planner, and route planner.

The route actor can receive two types of messages. It can be told to plan a route from the current location to a specific goal location, or it can be told to plan a route from a specific start location to a specific goal location. The two situations were selected based upon the way the mission planner formulates its plans and they will be covered in more detail in Chapter V. If the route planner is to plan a route from the current location to a specific goal location, it first must determine its current location. The route planner does this by checking the blackboard to find the current state of the vehicle. Once it knows its current location, it uses this information along with the goal location to satisfy conditional statements. If a route exists, then it is stored in the blackboard, otherwise a task failure is reported. The second type of message is handled in much the same way by the route planner except there is no need to initially

check the blackboard prior to searching the route database.

Obviously, this type of implementation can lead to a large database for even a small world model. The number of routes required to handle all possibilities in an environment with only 5 objects is 20 routes, and in general for an environment with n objects, $n(n-1)$ routes are required. Again, this only allows direct travel between objects. The available routes in this program will be looked at in the discussion of the world model.

The final actor grouped under the heading of main processor is the mission-planner actor. The mission-planner actor is more of an interface between the ROSS environment and the planning mechanism, and it merely consists of one message template. The mission-planner is invoked by the scheduler with a command directing it to plan for the current policy in the blackboard. The mission-planner actor then retrieves the current policy from the blackboard and passes it to the planning mechanism. Once the planner has finished its task, it then stores the results in the blackboard under the agenda attribute. Control is then returned to the scheduler and processing of the agenda begins in the plan projection phase.

The Sensor Processors. The sensor processors are shown in Figure 6 as three actors which interact with the scheduler, motor, and blackboard. The sensors modelled are: a gyro-compass, three sonars, and an optical shaft encoder.

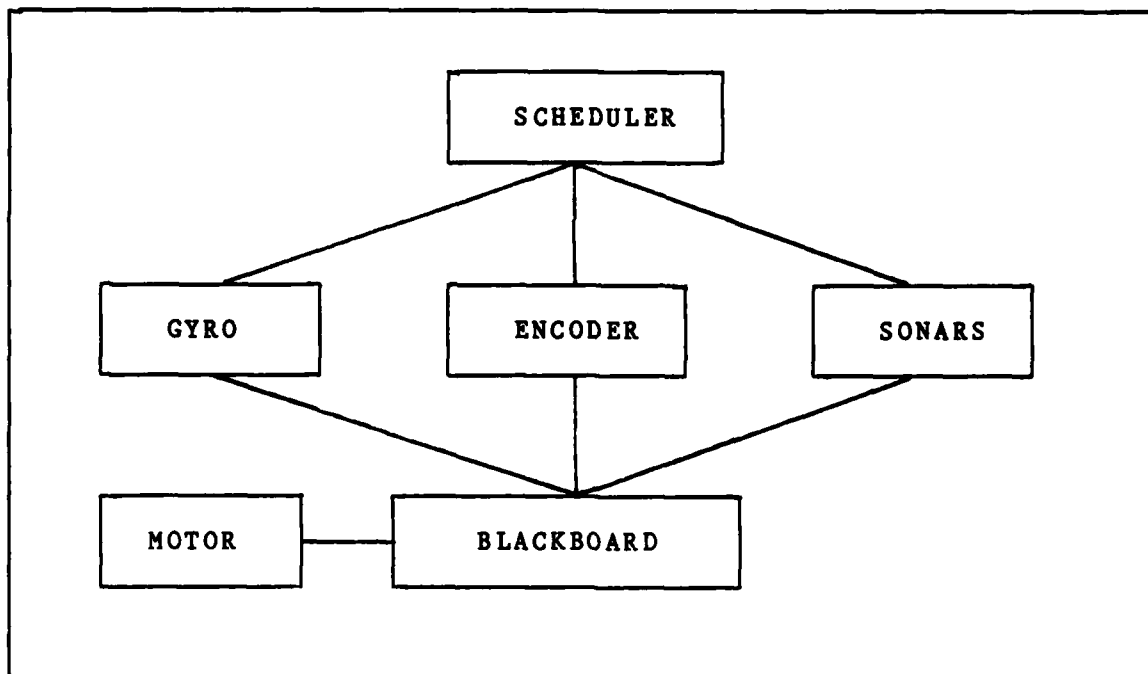


Figure 6. Communication Among Sensor Actors.

The gyro actor provides the vehicle with heading information which it stores in the blackboard for use by other routines in the system. The sonars provide object detection and ranging information for use by an object avoidance algorithm. Currently, the sonar information is not used in the program; however, future upgrades of the program involving a more elaborate world model description will make use of this information. Finally, the optical shaft encoder provides distance travelled information for use in computing the current coordinates of the vehicle.

The actors write to the blackboard by first issuing a request to the scheduler for access to the bus. When access is given, the actor writes to the blackboard and then notifies the scheduler when it is done. Each remaining actor repeats the

process until all sensor actors have written to the blackboard. When this operation is completed, the scheduler then regains control of the bus and invokes other actors to process the new information.

The Blackboard. The blackboard, shown in Figure 7, is a collection of actors which simulates a partitioned shared common memory allowing the main processor actors access to vital information. The blackboard contains time varying and static data that allows the main processor actors to monitor the state of the vehicle while it is in motion or stationary. This technique reduces the need for the main processor actors to directly query separate sensor processors for their information, thus reducing circuit complexity by using one clearinghouse for information. The blackboard implemented contains the minimal amount of information necessary to allow the vehicle to navigate through its environment. Subsequent upgrades of this system will incorporate the planning knowledge in the blackboard, but for the present implementation only data useful for navigation and execution monitoring is contained in the blackboard. However, a first step has been taken in this direction with the agenda and plan policy partitions.

The first partition in the blackboard hierarchy is the agenda actor. The agenda actor has an agenda-list attribute that contains a series of messages generated by the mission planner. The agenda-list functions in much the same fashion as a stack with messages being executed and then deleted from the top of the stack. The agenda-list is added to the blackboard by the mission

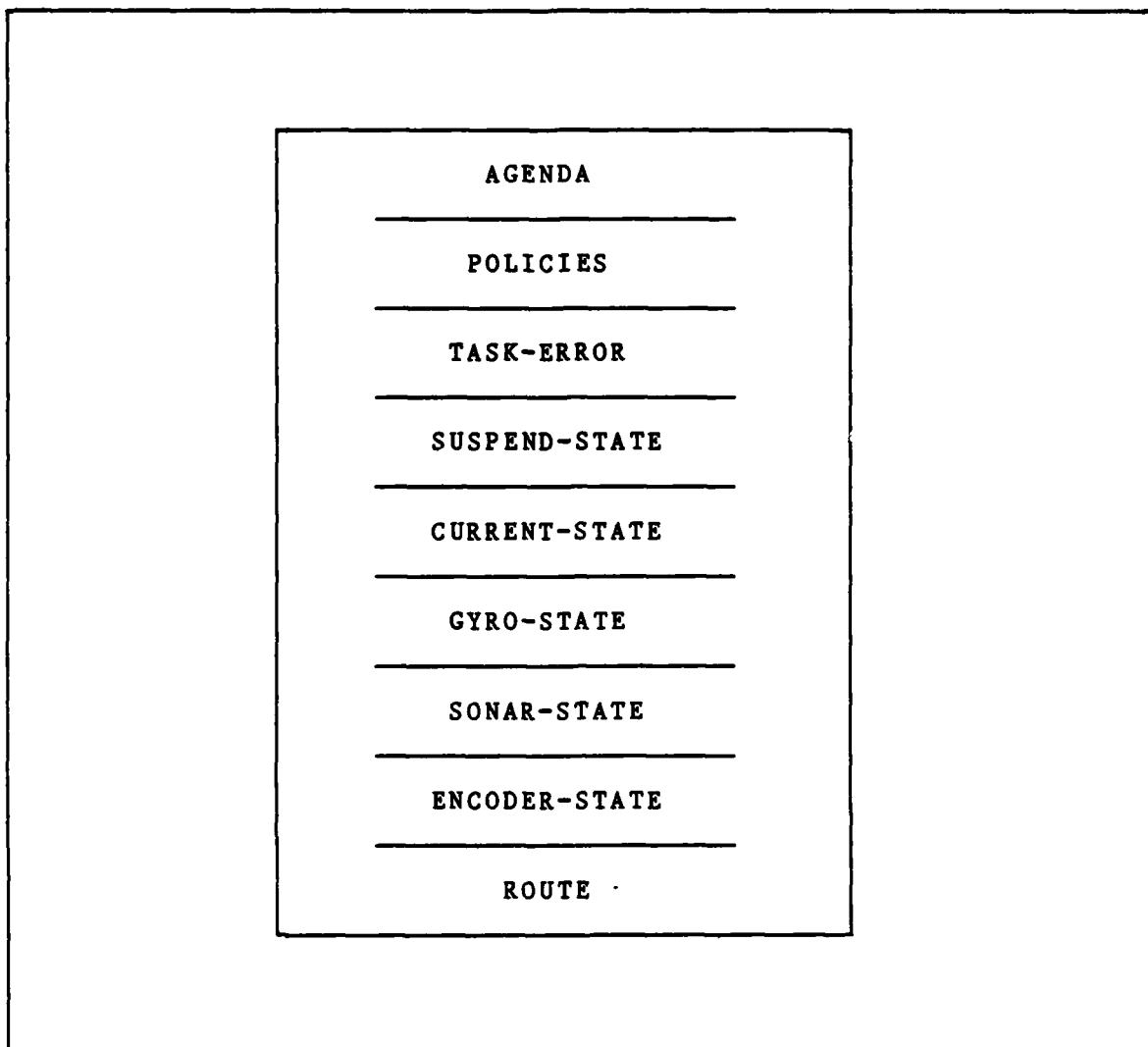


Figure 7. Blackboard Partitions.

planner and is read from the blackboard by the scheduler. This technique is similar to the one used in the Hayes-Roth (8) opportunistic planning model with the messages representing knowledge specialists or actors most appropriate to perform some task.

Another similarity to the Hayes-Roth model is seen in the

policy partition. This partition stores information concerning desirable attributes of a plan. The scheduler formulates its policy based on external inputs from a supervisor, or based on current conditions in the vehicle's environment. The scheduler then stores this policy in the blackboard and invokes the mission planner actor. The mission planner then tries to formulate a plan based on the current policy in the blackboard. Future upgrades of the system would incorporate the mission planner's knowledge base about policies in this partition.

The task-error partition contains information concerning tasks or actions that have not succeeded. So for example, if during plan projection the operating system discovers that a certain item crucial to the outcome of a plan is not available, a pointer to the failed plan would be stored in this partition. Thus, the scheduler could then access this partition and use the information while formulating a new plan policy.

The suspend-state partition contains information about the vehicle's last location prior to suspending a task. Therefore, if there is a need to stop in the middle of a task and start a new task, the vehicle can resume the old task at the point at which it left off upon completion of the new task. The suspend-state actor only need contain information about the location since the mission planner can direct the scheduler on how to recover any other information it may need.

The idea of suspending action and taking up a new task was once again influenced by the Hayes-Roth model of opportunistic planning in which a subject might discover an opportunity to perform a more important task while in the act of performing a

separate task. Thus, the vehicle might "realize" an opportunity to throw a switch that is near its current location, and hence accomplish a task that is not related to the plan in progress. An example such as this requires a much more sophisticated world model than the one implemented here, but it helps clarify the reasoning behind the partition.

The current-state actor stores information on the current condition of the vehicle such as its present location and available fuel. The present location of the vehicle is stored in x and y coordinate values under the attribute "coordinates," and the location of the vehicle is stored under the attribute "location". The amount of available fuel on the vehicle is stored under the attribute "resources," and it is updated each time the motor actor is issued a move command. The gyro-state, sonar-state, and encoder-state actors are locations where the corresponding sensor actors store their readings. Finally, the route actor is used to store data on the planned route of the vehicle. The planned route is stored as an A-list with each member of the list corresponding to a segment of the overall route. The planned route was stored in this manner to facilitate upgrade of the system to include objects in the environment. Since the objects would be used as turning points, each route segment indicates a portion of the path where an object is expected at the end of the segment. Also included as an attribute field of the route actor is the goal-state. The goal-state is the end point of the planned route and is provided separately in the event the planned route is altered.

The World Model

The world model implemented in the program was sufficient to demonstrate the concept of an operating system and mission planning mechanism working in conjunction. The vehicle's environment, shown in Figure 8, is modelled in a cartesian coordinate plane with eight objects identified by their x and y values. No attempt was made to model the objects in space, instead they are just considered points in the plane. A route between two objects is broken down into segments of unity distance and stored in an association list (A-list) with information on the route segment number, starting coordinates of the segment, length of the segment, and the heading. The format in a Lisp construct is as follows:

```
(route segment ((coordinates) length heading))
```

For example, a route between the base and the workbench might follow the coordinates: (0 0), (0 1), (0 2), (1 2), and (2 2). This route would be represented in an A-list as follows:

```
(setq *base-to-workbench*  
  '((0 ((0 0) 1 90)) (1 ((0 1) 1 90))  
    (2 ((0 2) 1 0)) (3 ((1 2) 1 0))))
```

Route segment zero is indicated by the first number followed by its starting coordinates, length of the vector, and heading. The route described by the above construct is depicted in Figure 9.

As mentioned earlier, an environment containing n objects would require $n(n-1)$ route descriptions to handle all

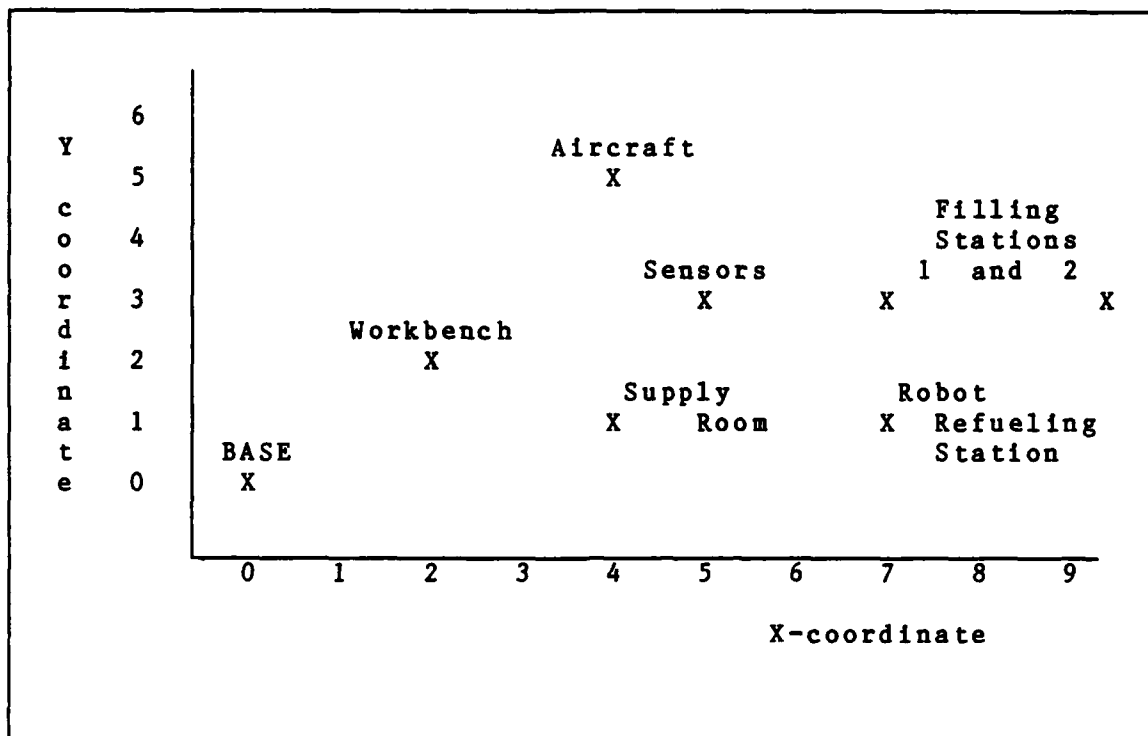


Figure 8. World Model of Autonomous Vehicle's Environment.

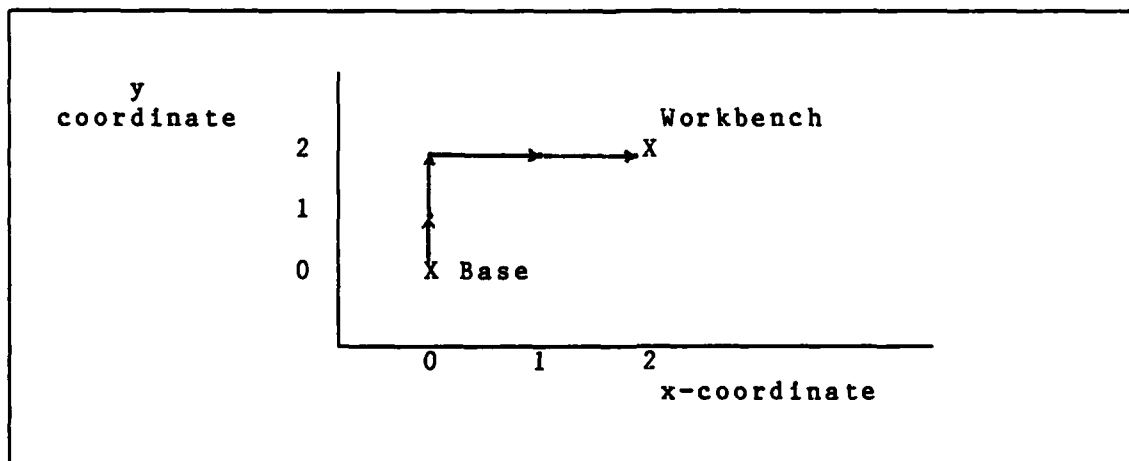


Figure 9. Base to Workbench Route.

possibilities for this type of explicit encoding. Thus, fifty-six routes would need to be stored in the database for this world model in order to provide access to all objects from any one object. However, only a small subset of these possibilities was selected in order to demonstrate carrying out four tasks. The routes encoded in the database are:

1. Base to workbench
2. Base to filling station
3. Workbench to aircraft
4. Aircraft to workbench
5. Workbench to filling station
6. Filling station to filling station 2
7. Filling station to robot refueling station
8. Robot refueling station to filling station
9. Workbench to sensors
10. Sensors to supply room
11. Supply room to workbench
12. Sensors to base
13. Aircraft to base

The routes were sufficient to demonstrate executing a series of plans and returning to the base location. Furthermore, an object map function provided the operating system with the ability to determine its current location based on the present x and y coordinate values. So, for example if the present x and y coordinates were 2.0 and 2.0, then the object map function would return the value "workbench". In this manner, a literal value as

well as a numeric value characterizes the description of the vehicle's current location. This method enables the formulation of message patterns to the route actor that are easier for the programmer to understand, and hence they increase the understandability of the ROSS code.

V. Planning System Design

Introduction

The top-down approach in problem solving has long been in favor in the engineering and computer science disciplines. For any complicated problem domain, the approach makes the problem more tractable by decomposing it into smaller subproblems. However, many times it is more beneficial to combine a bottom-up with a top-down approach in the problem solving process. By considering lower-level details in the design process, faulty reasoning or overly optimistic requirements can be eliminated early in the planning phase. Thus, a requirement to design a computational device with a certain clock speed might be relaxed after consideration of the available chip technology. Indeed, in many instances this top-down and bottom-up approach is an iterative process that may be repeated many times throughout the project's life cycle.

As discussed in Chapter III, humans use this method regularly in planning for a day's activities. They are flexible in that they can plan at a high level of abstraction while considering lower level details. Simulating this method in software is difficult at best, but it is crucial to providing an autonomous vehicle with the capability to work in a dynamic environment. Strictly adhering to the hierarchical planning approach discussed in Chapter III may lead to a very efficient plan; however, it may

also lead to repeated backtracking from the implementation level. The vehicle may have decided to construct a plan that involves using a particular object only to discover late in the planning phase that object is unavailable. Thus, it has to backtrack to the point at which it made that decision and replan with a new object. It can be argued, however, that a good hierarchical planner would not even consider deciding to use a particular object until it had formulated a very general plan. This is true; however, like the human planner, knowledge of lower level details can help reduce the planning time. Thus, the vehicle may be trying to formulate a plan to refuel an aircraft only to find there is no more fuel or all the nozzles are in use. Its knowledge of the state of the world model can be used in formulating the hierarchical plan.

The planning system described in this Chapter incorporates these concepts. It formulates plans using a hierarchy of representations combined with the detailed knowledge of the state of the world. Thus, the planning process would cease early on if no fuel was available, and instead, efforts would go into obtaining more fuel before planning for the original task resumes.

So far, attention has been focused on the higher levels in the hierarchy of the planning process and little mention has been given to the lower levels. In many cases, these lower level details are actions that occur frequently in a variety of different tasks. A vehicle may be required to use its arm in refueling an aircraft as well as in repairing the aircraft. Should it have to plan each time it is required to move its arm, or can this action be pre-planned and available at all times?

Humans learn to walk at an early age and thereafter never have to give conscious consideration to the act again. It would be an incredible burden if a person had to plan his motions each time he had to walk. Likewise, an autonomous vehicle would be needlessly burdened with the task of planning each time it had to use its arm. Instead, these plans can be formulated in advance and stored in a database where they are available when needed. This type of knowledge structure is known as a script, and it is useful in describing common sequences of events (13:203). Moreover, it has the additional advantage of being declarative in nature: the knowledge is explicit and need only be stored once. This is in agreement with the knowledge structure adopted for the more general planning and meta-planning rules. The knowledge structure is explicitly encoded as a static collection of facts that can be used both in understanding and planning.

There are several declarative mechanisms for representing knowledge and each have an advantage in a particular domain. In Wilensky's work on meta-planning, he was concerned with the domain of natural language text understanding. The influence of his work on this thesis has led to the adoption of a common knowledge representation. Furthermore, the choice of representation Wilensky used provides a useful structure for any natural language interface with an autonomous vehicle and a human supervisor.

Conceptual Dependency

A conceptual dependency (CD) structure represents relation-

ships among the components of an action. Conceptual dependency theory arose from the need to understand the meaning of sentences in natural language. The representation facilitates drawing inferences from the sentences by showing the relationships among the different components of the sentence. Furthermore, it is independent of the language in which the sentences were originally stated (13:222). Thus, two different people may describe an event using different words, but the CD representation may be the same. A CD represents the event underlying the sentences rather than representing the sentences themselves (14:13).

Whether the event is a physical or mental event, CD uses a simple structure to represent the core of an event. Every event represented in a CD structure has an actor, an action performed by the actor, an object that the action is performed upon, and a direction in which the action is oriented. Describing the relationships among these components is a set of primitive actions. Schank (14:17-25) describes eleven such primitive actions used in natural language understanding; however, this thesis focuses on a very small subset. Specifically, we are interested in the primitive acts move and grasp. These two acts are used repeatedly by the autonomous vehicle in the aircraft flightline domain. For example, a CD representation for the statement "The robot went to the aircraft" has the following components:

actor: robot
action: move
object: robot

direction: to aircraft
from current location

These four components adequately convey the meaning of the sentence, and they are in a form that suggests a software representation. Each component consists of a pair of attributes: a role attribute, and a filler. Thus, in the previous example the role attribute is "actor" and its filler is "robot". This construct was used in the planning system to represent the vehicles meta-planning, and general planning knowledge. However, much more freedom was taken in the use of primitive actions.

Schank describes a canonical method for using primitive acts in constructing CD's for natural language understanding. Strict adherence to these methods was not followed here because they tended to detract from the easy understanding of the knowledge rule base. Instead, great freedom was taken with CD theory in an effort to construct planning rules in a format that was easy for the reader to understand, while still maintaining a format that would allow future modifications to conform to the CD format described by Schank.

As mentioned earlier, the construct used was that of a role attribute and filler attribute. This role-pair construct is prefaced by a predicate which describes the event. The format is then:

(<predicate> <role-pair> <role-pair> ...)

Furthermore, these constructs can be nested so that the filler attribute can be a predicate-role-pair construct. For example, given the following sentence:

The robot should use its move-plan to
get from the workbench to the aircraft

A possible CD representation could be:

```
(move-plan (actor robot)
            (objective
              (prox (from workbench)
                    (to aircraft))))
```

The predicate here is "move-plan" which describes the subsequent event in the construct. The first role-pair is "actor robot" which tells us who is to do the event. The second role-pair is actually a nested role-pair. The role is "objective" which tells us this is what the robot intends to do, and its filler is another predicate-role-pair. The predicate here is "prox" which describes the subsequent role-pairs.

Hence, we can say the robot's objective is to place itself in the proximity of the aircraft by using its move-plan from the workbench. Schank, on the other hand, would represent this sentence in a more cryptic format that requires a deeper knowledge of CD theory in order to understand. The choice of predicates and role-pairs used in this thesis was based on the desire to allow the reader to immediately understand the rule base without requiring him to digress into a study of CD theory. Therefore, predicates such as "accomplish" and "do" are frequently used in the rule base.

The predicates used in the rule base also describe the type of rule, as well as the event itself. The predicate "action" signifies that the planning system is to formulate a plan to

accomplish this action. In this manner, the planning system can distinguish between the tasks it is to plan for and the plans for the task. So, if we wanted to give an autonomous vehicle the task of refueling an aircraft, we might formulate the task as follows:

```
(action (actor robot)
        (objective
          (refuel (object aircraft-1)
                  (with nozzle-1))))
```

Here we are telling the robot to have the objective of refueling aircraft-1 with nozzle-1. The planning system can then proceed with searching its knowledge base to construct a plan for this task.

There are other predicates that signify the type of rule, but these predicates need to be discussed in the context of the planning system. Therefore, they will be discussed in the section on program structure and briefly in the next section as we look at a primitive implementation of Wilensky's PAM program which served as a foundation for the planning system developed here.

Micro-PAM

Micro-PAM is a simple program that captures the essential flavor of Wilensky's PAM program (14:180; 15). The program is a story understander that uses knowledge about the planning process in order to explain a story. The rules it uses are declarative patterns that encode facts about how goals may give rise to plans. The program processes a story a sentence at a time making

inferences about the sentences by using its rule base. The inferences are retained and said to be predicted when some occurrence in the story logically follows from a previous occurrence. As an example, consider the following two line story:

John was hungry. John went to a restaurant.

Micro-PAM begins the understanding process by making inferences about the first sentence. In its knowledge base it may have the rule that says someone who is hungry may have the goal of obtaining food; however, this goal is not predicted since we have only processed the first sentence we cannot conclude that this is John's goal. Therefore, we discard this inference and place the first sentence under the category of theme. In other words, John's hunger is now the theme of this story, and we can relate any future inferences to this theme.

When the second sentence is processed, Micro-PAM may have in its rule base a rule that states anyone who goes to a restaurant may have the goal of obtaining food. This inference is still not predicted since we need to know that being hungry is instrumental to having the goal of obtaining food. Micro-PAM, however, may have in its rule base the rule that says a subgoal of obtaining food is to be hungry. Indeed, this subgoal is predicted by the theme of the story: John was hungry. Hence, these inferences have been predicted, and we can conclude that John's reason for going to the restaurant was to obtain food.

It is this mechanism of relating themes to inferences that is of interest here. The domains are quite different, but the mechanism is useful if we draw parallels between understanding a

story and planning for a task. In story understanding, we are trying to relate known themes to inferences, and we do not conclude any inferences until we can make such a relation. Likewise, in planning for some given task, we could make inferences about what needs to be accomplished, but we cannot conclude these inferences until there is a relation between the inferences and the statement of the task. Thus, the task itself becomes the theme of the planning process, and we try to relate inferences made to this theme. A success means we have not only constructed a plan, but we have understood the task. Using the form of conceptual dependency discussed previously, we might want to construct a plan for the following task:

```
(action (actor robot)
        (objective
         (refuel (object aircraft-1)
                  (with nozzle-1))))
```

This task then becomes the overall theme for our planning process, and inferences are made based on this theme. So, an inference might be made that says a refuel-plan is called for in this situation. This could be said to be related to the theme of refueling an aircraft, but we are trying to relate an inference to the more abstract notion of doing an action with something, in this case nozzle-1. Therefore, another inference could be made:

```
(move-plan (actor robot)
            (objective
             (prox (from filling-station)
                   (to aircraft)
                   (with nozzle-1))))
```

This inference says a move-plan should be used to get the robot

from the filling-station to the aircraft with nozzle-1. Again, we could say this inference relates to the theme, and indeed it does, except that the robot does not yet have a way to get nozzle-1. For that matter, it does not even have a way to get to the filling-station. We cannot say that this event is predicted by the theme because of the current state of the robot's world model; it needs to get to the workbench before it can get nozzle-1. Therefore, it continues making inferences as long as it has appropriate rules in its knowledge base. The next inference might be:

```
(grasp-plan (actor robot)
            (object nozzle-1)
            (location filling-station))
```

The robot now has nozzle-1, but there is still an inconsistency in its environment since its current location is different from the filling-station. The next inference would remedy this inconsistency:

```
(move-plan (actor robot)
            (objective
              (prox (to filling-station)
                    (from current-location)
                    (with nothing)))))
```

The key elements of this inference are that it moves the robot from its current location to the filling-station, and it requires that the robot have nothing in its grasp. This inference would be consistent with its current state since the very general term current-location is used, and the robot is not currently holding anything. Therefore, we can say that this inference was

predicted, and hence all the inferences were predicted. If the robot had run out of matching rules in its knowledge base before making a prediction, we could say that the robot did not know how to perform this task.

In general, an inference is predicted if it involves an action that is consistent with the vehicle's world model and the description of the task. Therefore, we are not really using the task itself to predict the inferences, but instead we are using a combination of the task and the vehicle's current state. The task is used immediately to infer the high-level solution of using the refuel-plan, whereupon the task becomes the overall theme to judge the validity of the intervening steps. This is in contrast to Micro-PAM which only allows a story's sentence to become a theme when it can no longer make any more inferences.

There is an advantage to this particular approach when one considers the problem of an autonomous vehicle attempting to understand the actions of another autonomous vehicle. By observing the actions of the second vehicle, the first vehicle can search its knowledge base to infer the second vehicle's task. If one vehicle has observed another moving to the filling station with nothing in its grasp, then the first vehicle can say this action would be predicted from an order to refuel an aircraft. With this assumption, the first vehicle can formulate an entire plan the second vehicle might follow and try to combine the actions of the two vehicles if they have movements in common. An attempt to combine plans with another vehicle might arise if one vehicle has some task that it is unable to accomplish alone. Combining plans is an acceptable strategy that humans use

regularly, and it is a form of meta-planning that Wilensky refers to as goal concordance, or a positive interaction between the goals of different planners (16:220). Obviously, this type of planning would require knowledge about the tasks of other vehicles in the environment and knowledge about how to plan for them. The structure of the rules in Micro-PAM's knowledge base, however, allows separate databases to be stored.

Micro-PAM's rules are grouped under four categories:

1. Instantiation
2. Planfor
3. Subgoal
4. Initiate

An instantiation relates events to plans they may be a part of. Therefore, an instantiation rule would relate an order to refuel an aircraft to the vehicle's refuel-plan. A planfor, on the other hand, relates plans to goals that might be applicable. So, a refuel-plan might be related to the goal of moving to the aircraft. The subgoal category would relate this goal to an applicable plan, such as the vehicle's move-plan. Finally, the initiate category relates the themes to goals or plans that the vehicle may infer. Hence, the initiate rules might relate a refuel order to a plan involving moving to a filling station.

The rules themselves are grouped under each category as pairs of CD structures such that the occurrence of one implies the other. The CD structures have specific predicates and roles; however, the fillers are variable patterns. Thus, the CD struc-

tures are very general and can be matched with an appropriate instance of the CD structure. For example, a very general form of the move-plan CD might be:

```
(move-plan (actor ?x)
            (objective
              (prox (to ?y)
                    (from ?z)
                    (with ?w))))
```

The variables in the CD are prefixed with a question mark which allows the pattern matcher to replace them with specific instances. So, ?x might be replaced with "robot" and so on.

The concept of making inferences and then trying to predict them is a component of Micro-PAM that was retained in this implementation. Micro-PAM has several drawbacks, however, that limit its usefulness as a planning mechanism. It has a fixed control structure that is really not conducive to implementing the concepts Wilensky discusses in his theory on meta-planning. Moreover, none of the rules in the knowledge base are indexed, so the program does a sequential search through the rule base until it finds a rule that matches. Also, Micro-PAM has no way of handling sophisticated goal relationships such as might occur when trying to combine plans. And finally, Micro-PAM does not really demonstrate the ability to use knowledge about the planning process while constructing plans. Furthermore, it has none of the other components Wilensky proposes, such as a plan projector and goal detector. Each of these drawbacks, however, were addressed in this thesis effort.

Program Structure

In order to reduce the time required for Micro-PAM to search its knowledge base, a discrimination net was used in order to index the rules. A discrimination net for indexing structures containing variables was implemented using the description in Charniak (2:162-169). The net facilitates the search through the knowledge base by returning the structures whose index matches an input. Thus, if the input is a CD that directs the vehicle to refuel an aircraft, the discrimination net is searched using this input to match the indexes in the net. When a match occurs, the most appropriate structure is returned, in this case a refuel-plan CD would be returned. Hence, the search is a best first search as opposed to Micro-PAM's depth first search. This type of indexing scheme is crucial for an application that contains a very large rule base, and indeed an autonomous vehicle would require a very large rule base for the type of planning described by Wilensky. Furthermore, meta-planning requires the capability to manipulate plans by combining, deleting, or altering the original scenarios. Therefore, some method of associating tasks with their plans is needed in order to implement aspects of meta-planning.

As mentioned in Chapter III, an acceptable meta-plan is to combine plans. But, what happens if two plans are combined and something occurs in the vehicle's environment that invalidates one of the plans? An occurrence such as this might require removing the affected plan. Therefore, we need a method of identifying the affected elements of the plan in the database.

Charniak describes a simplified version of Doyle's truth maintenance system that proves adequate for this application (2:193-201).

When plans are created, each element of the plan is stored in a structure with three components: the plan itself, a pointer indicating the task that it is a component of, and a pointer to the next element in the plan. Figure 10 shows an example of such a structure for the task of refueling an aircraft. Thus, if two plans have been combined, each component of the plan identifies its parent task and the next component in the plan. If one plan has to be deleted, common elements of the combined plans would not be deleted since they are justified by more than one task as shown by the pointer m003. This element of the plan is justified by two tasks, a001 and a002. Therefore, it would not be deleted if one of the tasks is invalidated. We now have a structure for the database that allows the application of meta-planning theory. However, the basic structure of Micro-PAM's knowledge base needs to be altered to separate the planning rules from the meta-planning rules. This type of division implies that two types of planning will occur: meta-planning and general planning. The basic inferencing mechanism, however, can be used for both types of planning if a common structure for the meta-planning and general planning rules is used.

Recall that the rules were grouped in Micro-PAM under four categories. The categories classified the type of rule contained as either an instantiation, planfor, subgoal, or initiate rule. This type of classification can be used to add in another type of rule -- a meta-planfor rule. A meta-planfor rule would relate

```

a001 ---> (refuel task)          a002 ---> (repair task)

m001 ---> (move-plan (a001) (m002))
m002 ---> (grasp-plan (a001) (m003))
m003 ---> (move-plan (a001 a002) (m004))
m004 ---> (refuel-plan (a001) (m005))
m005 ---> (repair-plan (a002) nil)
          .
          .
          .

```

Figure 10. Truth Maintenance.

policies about the plan to meta-plans for accomplishing those policies. If the current policy governing a series of tasks was to plan for those tasks as efficiently as possible, then a possible meta-plan to accomplish this might be to plan each task separately and then combine the plans. This would be represented in the knowledge base as follows:

```

(meta-planfor
  (accomplish (plan ?y)
              (combine ?y))
  (policy (planner ?x)
           (objective (plan
                       (efficiently ?y)))))

```

The predicate "accomplish" illustrates the freedom taken with CD structures. This predicate is not explicitly allowed in CD theory; however, it is more descriptive for this application and

was used in order to enhance the readability of the rule base.

The previous example illustrates another enhancement of the original Micro-PAM. The meta-plan rule tells the planner to plan for each task and then combine the resulting plans. This implies that the control structure of the planner is directed by the knowledge base dependent on the prevailing plan policy. Therefore, a different policy might involve using different components of the planner, and indeed this does occur. The planner is passed the current plan policy from the operating system, whereupon the meta-plan rule base is used to specify an initial flow of control. After that, the planner is free to alter the policy based upon its ability to plan for a particular task.

An autonomous vehicle could be given an order to plan for a particular task such as a refuel task. The planner would begin by initially planning for the policy in order to define the appropriate control structure. After obtaining the control flow, the planner begins processing the meta-plan which tells the planner which functions to use for the current policy. As in the previous example, this might involve planning for the task and then combining the plan with other plans. While planning for the task, the planner may be unable to predict any of the inferences because perhaps the vehicle's world model is inconsistent with the specification of the task. The task specification may have directed the vehicle to use nozzle-1, but the planner is unable to locate nozzle-1 because it may be in use. Therefore, we have a plan failure, and it is unreasonable for the planner to continue with the original control structure. However, an

acceptable meta-plan for this occurrence is to alter the scenario. In other words, if a planner is unable to accomplish an original task, then try changing some aspect of the specification so that the ultimate goal is still achieved. Therefore, the planner could alter the task specification by using nozzle-2 instead of nozzle-1. The ultimate goal of refueling the aircraft can still be accomplished because, in this case, it is irrelevant which nozzle is used.

The planner, however, needs to query the meta-plan knowledge base. Therefore, it uses the original task specification to see if any suitable meta-plan exists. The meta-plan would specify which functions to use in case no plan could be constructed for a task. So a suitable meta-plan for this example might be:

```
(accomplish (alternative-scenario task))
```

The meta-plan directs the planner to use the function alternative-scenario with the original task. Planning can then proceed if an alternative scenario does exist.

In the event no alternative scenario exists, the entire planning process for the task has failed. The planner needs to formulate this into a policy identifying the function that failed rather than the task. Thus, the initial policy has been altered by the planner in an attempt to salvage the entire planning process. The new policy might take the form:

```
(policy (planner vehicle)
  (objective
    (plan
      (failed "function that failed")))))
```

This policy identifies which function in the planner has failed. Using this new policy, the planner can query the meta-plan rule base to see if another control flow might salvage the planning process. If no new control flow exists, then the planner may just have to abandon the task and try to plan for other more valuable tasks.

The preceeding description of the meta-plan rule base implies the existence of functions that can formulate policies, seek alternative scenarios, and combine plans. Each of these functions was added to the original Micro-PAM in order to provide the capability to accomplish meta-plans. Another function, not previously mentioned, was added in order to provide the function of abandoning plans. Furthermore, a sorting function was added to provide the capability of ordering the tasks by decreasing value.

Each plan in the rule base is given a plan value that denotes its relative importance. A refuel-plan might have a higher value than say a plan involving sweeping the floor. Hence, the planner has the capability to judge the importance of each task. Therefore, if a meta-plan calls for abandoning a task, a policy might be spawned that directs the planner to maximize the value of the remaining tasks. A suitable meta-plan to accomplish this is to sort the remaining tasks by value and then plan for them. Once planning has been accomplished, the planner can then proceed to the final phase -- refining the plans.

Up until now, components of a plan have only been stated in abstract terms such as refuel-plan, move-plan, and grasp-plan. The planner now needs to transform these components into actions

that the operating system can understand. This phase of planning has been termed the refinement phase, and it is accomplished by the refine function. Each plan in the rule base has associated with it a script, or a set of pre-defined actions for accomplishing the plan. The scripts are in the form of ROSS messages that can be used by the operating system to direct actions. For example, the grasp-plan might have the following script:

```
(tell arm move to location ?y)
(tell arm close hand)
(tell arm retract)
```

The script consists of three ROSS messages to the appropriate actor in the operating system. Furthermore, they are in a general format since the y variable is not specified. Therefore, it is the job of the refine function to fill in this variable before passing the script to the operating system. The messages are indeed actions that would be performed no matter what the vehicle had to grasp. The arm is first moved to some location, then the hand is closed, and finally the arm is retracted. The only variable in the action is the location. Moreover, it is not necessary to tell the operating system to first check for an empty hand since this is handled by the planner at an abstract level in the planning phase. The planner will not allow grasp-plans to occur in sequence without an intervening ungrasp-plan. Nevertheless, the operating system does indeed check the black-board to ensure no object is in the vehicle's grasp.

The incorporation of scripts required that a separate rule

base be created to store the scripts. Rather than just add an additional category, such as a script-for category, the scripts were placed in a separate rule base which allowed the use of the same categories but in a different context. The preceeding script for the grasp-plan could then be stored in this rule base, known as the refine-rules database, in the following form:

```
(planfor
  ((tell arm move location ?y)
   (tell arm close hand)
   (tell arm retract))
  (grasp-plan (planner ?x)
              (object ?y)
              (location ?w)))
```

This rule says that a plan for a grasp-plan is the script encoded in the rule. The refine function would have the job of finding the coordinates of object ?y and substituting it in the script. The decision to create a separate rule base was based upon the possibility of allowing the planning system to manipulate the scripts exclusively, rather than first plan at a high level of abstraction. If only some minor deficiency exists in a script, the planner could plan using the scripts themselves, rather than their more abstract equivalents. This, however, is a function that was not implemented. Nevertheless, a future upgrade could make use of the structuring of the rule base to incorporate such a capability.

To summarize, the basic inferencing mechanism of Micro-PAM, along with its CD structure, was used as a foundation for the planning system in this thesis. Enhancements to the basic Micro-PAM structure include the following:

1. Discrimination net to index the rules
2. Basic truth maintenance capability
3. Meta-plan category
4. Functions to handle meta-planning
5. Separate rule base for scripts
6. Capability to judge value of plan
7. Flow of control specified in rule base

VI. Integration and Testing

Introduction

Although this thesis has been divided into operating system design and planning system design, the two components are in effect two planners with different philosophies. The operating system is a planner whose entire knowledge is procedurally encoded in each of the actors. Therefore, adding to its knowledge base requires adding new actors, or adding new behaviors to the existing actors. The mission planner's knowledge base, on the other hand, is more declarative in nature. Adding new planning knowledge means adding new rules, and adding new meta-plans simply means specifying a new control flow.

The dichotomy becomes less evident, however, when the two components are integrated. The output of the mission planner is now in effect determining the control flow of the operating system. The mission planner suggests how the operating system is to use its actors to accomplish a task. However, it does not fully specify the control flow because its knowledge of the operating system is limited. Likewise, the operating system's knowledge of the mission planner is limited to policies it knows the mission planner can handle. Thus, we have the interface points for the two components. The operating system must formulate its commands into policies the mission planner can understand, and the mission planner must formulate its commands into

messages the operating system can understand. The result of the integration illustrates the clearly defined jobs of each component.

Integration

The interface points have already been identified in Chapters IV and V. The operating system communicates with the mission planner through the use of the mission-planner actor. As mentioned in Chapter IV, the mission-planner actor only has one message template and behavior. Its sole job is to obtain the current policy from the blackboard and to invoke the planning function. Once planning has ended, the mission planner actor stores the resulting plans in the blackboard and returns control to the scheduler. Thus, the interface in one direction is straightforward and provides great flexibility for future upgrade.

The planning system communicates with the operating system in a like fashion through the use of scripts. The scripts are in reality ROSS messages in CD forms. Since the operating system cannot understand CD structures, it needs to translate them into the proper format. The nature of Lisp, however, allows this to be accomplished quite easily. CD forms are just nested lists, and in order to translate them into the proper format, the operating system just needs to remove all of the internal parentheses.

Testing

Testing was done on a VAX 11/780 running under the UNIX

operating system. The objectives of the test were to demonstrate:

1. Planning and meta-planning
2. Plan projection
3. Goal detection

Four tasks were selected to demonstrate these objectives, as well as the meta-planning concepts of:

1. Combining plans
2. Abandoning plans
3. Seeking alternative scenarios
4. Recovering from plan failure

The four tasks selected were stored under the global variable robot-task in the CD form:

```
(setq robot-task
  '((action (actor (vehicle (name (robot-1))))
    (refuel aircraft)
    (with nozzle-1))
    (action (actor (vehicle (name (robot-1))))
    (repair engine)
    (location aircraft)
    (with engine-tools))
    (action (actor (vehicle (name (robot-1))))
    (maintain work-bench)
    (with work-bench-supplies))
    (action (actor (vehicle (name (robot-1))))
    (maintain sensors)
    (with sensor-supplies))))
```

The first task directs the vehicle to refuel an aircraft with a nozzle, while the second task directs the vehicle to repair the engine on the same aircraft. The next two tasks are less important general house-keeping functions to be performed by the

vehicle. It must ensure the work-bench has ample supplies and the various sensors in the area are functioning properly. The vehicle's environment is the same as in Figure 8, repeated here in Figure 11.

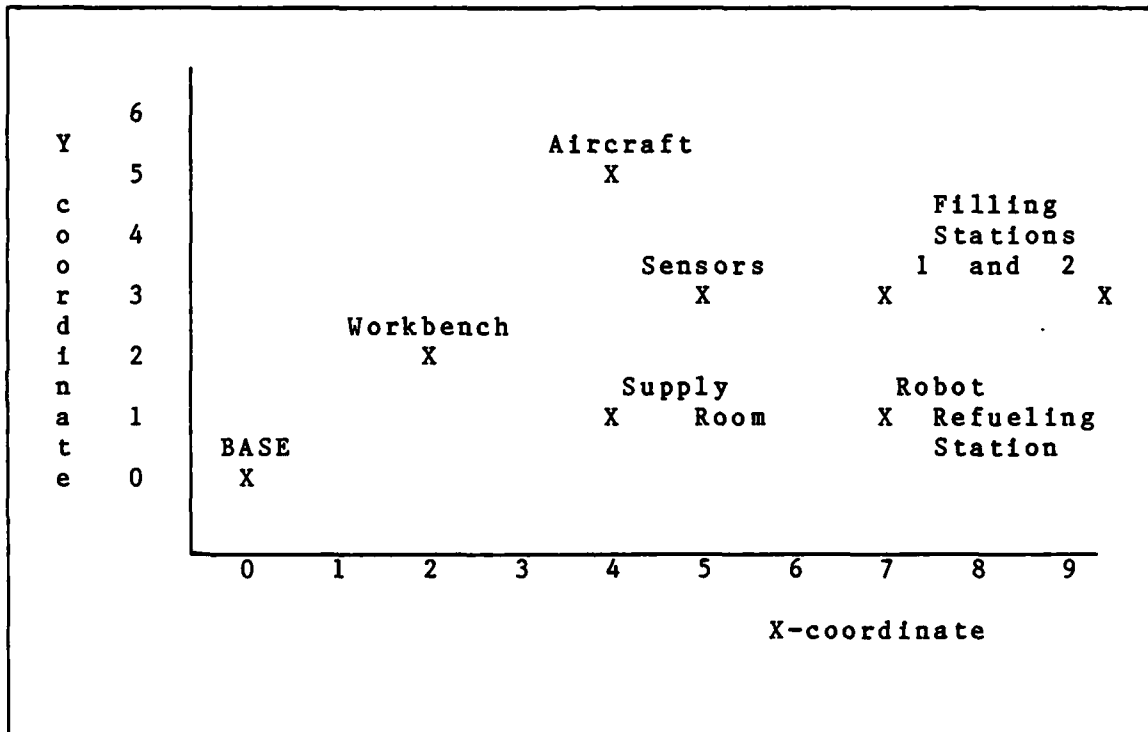


Figure 11. Autonomous Vehicle's Environment.

The following is an edited transcript of the test run. Comments have been included to highlight the important concepts. A complete transcript of the test run can be found in the Appendix.

===== ROSS =====

VERSION: Wed Jan 19 11:26:26 1983

The ROSS environment has been invoked and the user now asks the scheduler actor to run a simulation for a maximum of 300 time steps.

-> (ask scheduler go 300)

```
*** ROBOT SIMULATION ***
Current Coordinates: (0.0 0.0)
Current Location:    base
Current Resources:   100
```

After printing a banner showing its current state, the scheduler invokes the planner actor and tells it to plan. The planner signals that it is ready to plan for the current policy.

Ready to plan

Looking for a meta-plan for Policy:

```
(policy (planner robot)
  (objective (plan (efficiently robot-task))))
```

Possible explanation assuming

```
(accomplish (clear-globals all)
  (process-cds robot-task)
  (combine-plans *task-plans*)
  (process-cds *return-action*)
  (refine nil))
```

The planner has arrived at a meta-plan to accomplish the planning policy. First it is to clear any global variables, plan

for each task separately, combine the plans, plan a return action, and finally refine the plans into scripts.

A meta-plan for this policy is:

```
((clear-globals all) (process-cds robot-task)
  (combine-plans *task-plans*)
  (process-cds *return-action*)
  (refine nil))
```

It now plans for the first task.

Planning

```
(action (actor (vehicle (name (robot-1))))
  (refuel aircraft)
  (with nozzle-1))
```

Event not predicted

Possible explanation assuming

```
(refuel-plan (planner (vehicle (name (robot-1))))
  (object aircraft)
  (with nozzle-1))
```

Event not predicted

Possible explanation assuming

```
(goal (planner (vehicle (name (robot-1))))
  (objective
    (prox (actor (vehicle (name (robot-1))))
      (location aircraft)
      (with nozzle-1))))
```

Event not predicted

Possible explanation assuming

```
(move-plan (planner (vehicle (name (robot-1))))
  (from (prox (location nil)))
  (to (prox (location aircraft)))
  (with nozzle-1))
```

No usable inferences from

```
(refuel-plan (planner (vehicle (name (robot-1))))  
              (object aircraft)  
              (with nozzle-1))
```

No usable inferences from

```
(action (actor (vehicle (name (robot-1))))  
         (refuel aircraft)  
         (with nozzle-1))
```

No inference chain found--

The planner was unable to find a plan for the first action because nozzle-1 was unavailable. Therefore, it seeks a meta-plan for the task.

Seeking Meta-Plan for:

```
(action (actor (vehicle (name (robot-1))))  
         (refuel aircraft)  
         (with nozzle-1))
```

Possible explanation assuming

```
(accomplish  
(alternative-scenario  
  (action (actor (vehicle (name (robot-1))))  
           (refuel aircraft)  
           (with nozzle-1))))
```

Trying Alternative Scenario

Possible explanation assuming

```
(accomplish  
(justify-alternative  
  (action (actor (vehicle (name (robot-1))))  
           (refuel aircraft)  
           (with nozzle-2))))
```

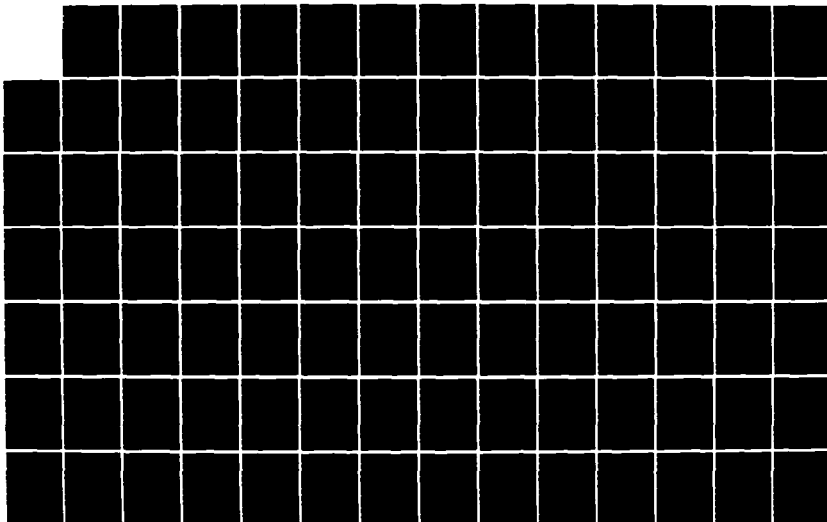
AD-A163 936

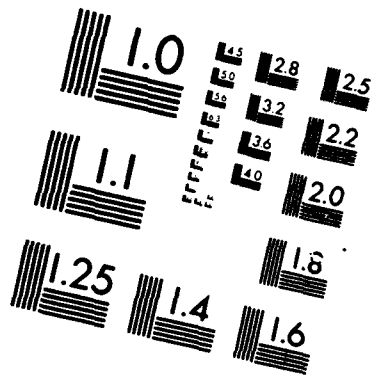
AUTONOMOUS VEHICLE MISSION PLANNING USING AI
(ARTIFICIAL INTELLIGENCE) TE. (U) AIR FORCE INST OF
TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.
S E STOCKBRIDGE DEC 85 AFIT/GE/ENG/85D-45 F/G 6/4

273

UNCLASSIFIED

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

The planner has arrived at the meta-plan of trying an alternative scenario for the task. An alternative scenario for this particular task is to try using a different nozzle to accomplish the task.

Planning

```
(action (actor (vehicle (name (robot-1))))  
        (refuel aircraft)  
        (with nozzle-2))
```

Event not predicted

Possible explanation assuming

```
(refuel-plan (planner (vehicle (name (robot-1))))  
             (object aircraft)  
             (with nozzle-2))
```

Event not predicted

Possible explanation assuming

```
(goal (planner (vehicle (name (robot-1))))  
      (objective  
        (prox (actor (vehicle (name (robot-1))))  
              (location aircraft)  
              (with nozzle-2))))
```

Event not predicted

Possible explanation assuming

```
(move-plan (planner (vehicle (name (robot-1))))  
           (from (prox (location filling-station)))  
           (to (prox (location aircraft)))  
           (with nozzle-2))
```

Event not predicted

Possible explanation assuming

```
(goal (planner (vehicle (name (robot-1))))  
      (objective  
        (grasp (object nozzle-2)  
              (location filling-station))))
```

Event not predicted

Possible explanation assuming

```
(grasp-plan (planner (vehicle (name (robot-1))))  
            (object nozzle-2)  
            (location filling-station))
```

Event not predicted

Possible explanation assuming

```
(goal (planner (vehicle (name (robot-1))))  
      (objective  
        (prox (actor (vehicle (name (robot-1))))  
              (location filling-station)  
              (with nil))))
```

Event not predicted

Possible explanation assuming

```
(move-plan (planner (vehicle (name (robot-1))))  
           (from (prox (location current-location)))  
           (to (prox (location filling-station)))  
           (with nil))
```

Event predicted from

```
(action (actor (vehicle (name (robot-1))))  
        (refuel aircraft)  
        (with nozzle-2))
```

Adding inference chain to data base

The planner has arrived at a plan for the first task and then proceeds to plan for the second task of repairing the aircraft's engine.

Planning

```
(action (actor (vehicle (name (robot-1))))  
        (repair engine)  
        (location aircraft)  
        (with engine-tools))
```

Event not predicted

Possible explanation assuming

```
(repair-engine-plan (planner (vehicle (name (robot-1))))  
  (object engine)  
  (location aircraft)  
  (with engine-tools))  
  .  
  .  
  .
```

Intervening planning steps have been left out, but the planner successfully arrives at a plan for the second task.

Event predicted from

```
(action (actor (vehicle (name (robot-1))))  
  (repair engine)  
  (location aircraft)  
  (with engine-tools))
```

Adding inference chain to data base

The planner now plans for the third task of maintaining the work-bench.

Planning

```
(action (actor (vehicle (name (robot-1))))  
  (maintain work-bench)  
  (with work-bench-supplies))
```

Event not predicted

Possible explanation assuming

```
(replenish-work-bench-plan  
  (planner (vehicle (name (robot-1))))  
    (maintain work-bench)  
    (with work-bench-supplies))  
  .  
  .
```

Intervening planning steps have once again been left out, but the planner successfully plans for the third task.

Event predicted from

```
(action (actor (vehicle (name (robot-1))))  
        (maintain work-bench)  
        (with work-bench-supplies))
```

Adding inference chain to data base

The planner now plans for the fourth and final task of maintaining the sensors in the work area.

Planning

```
(action (actor (vehicle (name (robot-1))))  
        (maintain sensors)  
        (with sensor-supplies))
```

Event not predicted

Possible explanation assuming

```
(sensor-repair-plan (planner (vehicle (name (robot-1))))  
                    (maintain sensors)  
                    (with sensor-supplies))
```

.
.
.

Event predicted from

```
(action (actor (vehicle (name (robot-1))))  
        (maintain sensors)  
        (with sensor-supplies))
```

Adding inference chain to data base

The planner is now finished with each task and proceeds to the task of combining the plans to accomplish the policy of creating an efficient plan.

Combining Plans

The planner has finished combining plans and now sorts the plans so that the most important plans are done first.

Sorting Plans by value

The planner now plans for a return action. Once the vehicle has accomplished all of its tasks, it will need some way to get back to its home base; therefore, the planner anticipates this and plans appropriately.

Planning

```
(action (actor (vehicle (name (robot-1))))  
        (objective (return (location base))))
```

Possible explanation assuming

```
(move-plan (planner (vehicle (name (robot-1))))  
            (from (prox (location current-location)))  
            (to (prox (location base)))  
            (with nil))
```

Event predicted from

```
(action (actor (vehicle (name (robot-1))))
```

```
(objective (return (location base))))
```

Adding inference chain to data base

The planner now refines the plans into scripts that the operating system can understand.

Refining the Plans

Refining

```
(move-plan (planner (vehicle (name (robot-1)))))  
  (from (prox (location current-location)))  
  (to (prox (location filling-station)))  
  (with nil))
```

Refining

```
(grasp-plan (planner (vehicle (name (robot-1)))))  
  (object nozzle-2)  
  (location filling-station))
```

Refining

```
(move-plan (planner (vehicle (name (robot-1)))))  
  (from (prox (location filling-station)))  
  (to (prox (location aircraft)))  
  (with nozzle-2))  
.  
.  
.
```

Once the planner has completed the refinement phase, it prints the database. A portion of the database is shown here as composed of scripts in the form of ROSS messages. Notice that each script is composed of a message, a pointer to its parent task, and a pointer to its subsequent task. The header "executive" in each message tells the operating system to use the

function executive to process the message. In this manner, the initial control flow of the operating system is specified.

The database now contains:

```
((executive (tell (route-planner))
              (start current-location)
              (goal filling-station)) (a00005)
                                         m00032)
((executive (tell (scheduler))
              (move route)) (a00005)
                                         nil)
((executive (tell (sensor))
              (locate nozzle-2)) (a00005)
                                         m00034)
.
.
.
```

The planning phase is now complete and control is returned to the operating system for the plan projection phase. The operating system then takes the script database and begins processing the scripts one at a time. The first script is a command to move to the filling-station.

Move Robot
Start Location: base
Destination: filling-station

Encoder initializes memory
Sonar initializes memory
Gyro initializes memory

**** Time step: 1 ****

Confirm initialization

Initialization Confirmed
Ready For Motion Simulation

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 2 ****

New coordinates are (0.5 0.0)

Robot moves forward: 0.5 meters

**** Time Step: 5 ****

New coordinates are (1.314835507758704 0.3884309501755995)

At this point while travelling to the filling-station the vehicle strays from its course. Therefore, the errorhandler actor is called to handle the error condition.

Course deviation -- calling errorhandler

New heading has been computed

Turn right -9.227130622162847 degrees
Reorient Robot -9.227130622162847 degrees

.
.
.

The vehicle eventually achieves its goal and then proceeds to accomplish the task of getting nozzle-2. However, the amount

of effort it took the vehicle to travel to the filling-station caused a low fuel state. Thus, a goal detection occurs and the operating system informs the planner of the new goal.

**** Time Step: 26 ****

New coordinates are (7.000796326710733 2.999999682931835)

Goal achieved

Remaining resources: 78
Current location: filling-station
Coordinates: (7.0 3.0)

Robot Fuel Critically Low

Looking for a meta-plan for Policy:

(policy (planner robot)
 (objective (plan (maximize *low-robot-fuel-task*))))

Possible explanation assuming

(accomplish (save-top-level-tasks nil)
 (clear-globals some)
 (process-cds *low-robot-fuel-task*)
 (process-cds *return-suspend-action*)
 (sort *task-plans*)
 (refine nil)
 (restore-top-level-tasks nil))

A meta-plan for this policy is:

((save-top-level-tasks nil) (clear-globals some)
 (process-cds
 low-robot-fuel-task)
 (process-cds
 return-suspend-action)
 (sort *task-plans*)
 (refine nil)
 (restore-top-level-tasks nil))

Planning

```
(action (actor (vehicle (name (robot-1))))  
        (correct (state (low robot-fuel))))
```

Event not predicted

Possible explanation assuming

```
(refuel-robot-plan (planner (vehicle (name (robot-1))))  
                   (correct (state (low robot-fuel)))  
                   (with suspended-state))  
.  
.  
.
```

Event predicted from

```
(action (actor (vehicle (name (robot-1))))  
        (correct (state (low robot-fuel))))
```

Adding inference chain to data base

Planning

```
(action (actor (vehicle (name (robot-1))))  
        (objective (return (location suspend-location))))  
.  
.  
.
```

Event predicted from

```
(action (actor (vehicle (name (robot-1))))  
        (objective (return (location suspend-location))))
```

Adding inference chain to data base

Sorting Plans by value

Refining the Plans

Refining

```
(suspend-state-plan (planner (vehicle (name (robot-1)))))
```

Refining

```
(move-plan (planner (vehicle (name (robot-1)))))  
  (from (prox (location suspend-location)))  
  (to (prox (location robot-fuel)))  
  (with suspended-state))
```

Refining

```
(refuel-robot-plan (planner (vehicle (name (robot-1)))))  
  (correct (state (low robot-fuel)))  
  (with suspended-state))  
.  
.  
.
```

The planner has successfully planned for the low fuel condition and control is returned to the operating system to continue with the plan projection phase. In order to prevent the operating system from having to start the plan projection phase all over again, the planner has instructed the operating system to suspend its current state until it refuels. Therefore, plan projection for the original tasks can resume at the point at which it was interrupted.

Move Robot

```
Start Location:  filling-station  
Destination:    robot-fuel
```

```
Encoder initializes memory  
Sonar initializes memory  
Gyro initializes memory
```

```
**** Time step: 27 ****
```

```
Confirm initialization
```

Heading error in initialization
Calling Error Handler

New heading has been computed

Turn right -180.0 degrees
Reorient Robot -180.0 degrees

Re-initialize memory

Encoder initializes memory
Sonar initializes memory
Gyro initializes memory

**** Time step: 28 ****

Confirm initialization

Initialization Confirmed
Ready For Motion Simulation

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

.
.
.

The vehicle eventually arrives at the robot refueling
station, and the operating system instructs the vehicle to refuel
by inserting its robot finger into the refuel socket.

**** Time Step: 32 ****

New coordinates are (7.001592653421467 1.000000634136331)

Goal achieved

Remaining resources: 73
Current location: robot-fuel
Coordinates: (7.0 1.0)

Sensors have located refuel-socket

Arm moved to refuel-socket

robot-finger has been inserted

Robot Refueled
Current resources: 1000

The vehicle now returns to the point at which it was interrupted.

Move Robot
Start Location: robot-fuel
Destination: filling-station

Encoder initializes memory
Sonar initializes memory
Gyro initializes memory

**** Time step: 33 ****

Confirm initialization

Heading error in initialization
Calling Error Handler

New heading has been computed

Turn left 180.0 degrees
Reorient Robot 180.0 degrees

Re-initialize memory

Encoder initializes memory
Sonar initializes memory
Gyro initializes memory

.
. .
.

After arriving back at the filling-station, the operating system continues the plan projection phase. However, in the time it took the robot to refuel itself, nozzle-2 has become unavailable. Therefore, the operating system detects a new goal for the vehicle and formulates it into a policy for the planning system. The situation is similar to what occurred with nozzle-1; however, this time the nozzle was unavailable during the plan projection phase.

**** Time Step: 38 ****

New coordinates are (7.001592653421467 2.999999365863669)

Goal achieved

Remaining resources: 995
Current location: filling-station
Coordinates: (7.0 3.0)

Sensors cannot Locate: nozzle-2

Task Failure Detected

Failed Function: (a00005)

Modifying Policy

The planner is now looking for a meta-plan for the failed task indicated by its pointer a00005. The solution is similar to

what occurred with nozzle-1 except this time the planner abandons the task a00005 and then seeks an alternative scenario. The intervening functions it performs all relate to management of the plan database.

Looking for a meta-plan for Policy:

```
(policy (planner robot)
        (objective (plan (failed a00005))))
```

Possible explanation assuming

```
(accomplish (abandon a00005)
            (move-to-top-level nil)
            (clear-globals some)
            (extern-plan-fail a00005)
            (sort *task-plans*)
            (push-new-plan nil)
            (refine nil))
```

A meta-plan for this policy is:

```
((abandon a00005) (move-to-top-level nil)
 (clear-globals some)
 (extern-plan-fail a00005)
 (sort *task-plans*)
 (push-new-plan nil)
 (refine nil))
```

Abandoning Plan a00005

Seeking Meta-Plan for:

```
(action (actor (vehicle (name (robot-1))))
        (refuel aircraft)
        (with nozzle-2))
```

Possible explanation assuming

```
(accomplish
  (alternative-scenario
    (action (actor (vehicle (name (robot-1))))
            (refuel aircraft))
```

(with nozzle-2))))

Trying Alternative Scenario

Possible explanation assuming

```
(accomplish
  (justify-alternative
    (action (actor (vehicle (name (robot-1))))
      (refuel aircraft)
      (with nozzle-3))))
```

Planning

```
(action (actor (vehicle (name (robot-1))))
  (refuel aircraft)
  (with nozzle-3))
```

Event not predicted

Possible explanation assuming

```
(refuel-plan (planner (vehicle (name (robot-1))))
  (object aircraft)
  (with nozzle-3))
```

.
.
.

The planner arrives at a solution and control is once again returned to the operating system for the plan projection phase. Nozzle-3, however, is located at filling-station-2 so the operating system instructs the vehicle to move to filling-station-2.

Move Robot

Start Location: filling-station
Destination: filling-station-2

Encoder initializes memory
Sonar initializes memory

Gyro initializes memory

**** Time step: 39 ****

Confirm initialization

Heading error in initialization
Calling Error Handler

New heading has been computed

Turn right -90.0 degrees
Reorient Robot -90.0 degrees

Re-initialize memory

Encoder initializes memory
Sonar initializes memory
Gyro initializes memory

**** Time step: 40 ****

Confirm initialization

Initialization Confirmed
Ready For Motion Simulation

Robot moves forward: 0.5 meters

.
.
.

The vehicle arrives at filling-station-2 and the operating system instructs the arm to grasp nozzle-3. Once nozzle-3 has been grasped, the vehicle then proceeds to the aircraft.

**** Time Step: 45 ****

New coordinates are (9.0 3.0)

Goal achieved

Remaining resources: 990
Current location: filling-station-2
Coordinates: (9.0 3.0)

Sensors have located nozzle-3

Hand opened

Arm moved to nozzle-3

Hand closed

Arm retracted

Move Robot

Start Location: filling-station-2
Destination: aircraft

.
.
.

The vehicle arrives at the aircraft and then proceeds to refuel the aircraft. Once refueling has been accomplished, the operating system then moves to the work-bench to gather tools to repair the engine on the aircraft. Notice that the plan of refueling the aircraft and repairing the engine have been combined in such a fashion that there is a smooth transition between the two plans. Once refueling is accomplished, the vehicle moves from its current location to the work-bench to carry out the next task. The plans could not be further combined, however, since each plan accomplishes a critical task and should not be interrupted.

**** Time Step: 67 ****

New coordinates are (4.00000126827246 5.001592652916487)

Goal achieved

Remaining resources: 974
Current location: aircraft
Coordinates: (4.0 5.0)

Sensors have located aircraft

Sensors have located filler-cap

Hand opened

Arm moved to filler-cap

Hand closed

Arm rotated

Arm retracted

nozzle-3 has been inserted

Move Robot

Start Location: aircraft
Destination: work-bench

.
.
.

**** Time Step: 83 ****

New coordinates are (2.000796326710733 2.000000317068165)

Goal achieved

Remaining resources: 961
Current location: work-bench
Coordinates: (2.0 2.0)

Sensors have located engine-tools

Hand opened

Arm moved to engine-tools

Hand closed

Arm retracted

Move Robot

Start Location: work-bench
Destination: aircraft

**** Time Step: 99 ****

New coordinates are (4.000796326710733 4.999999682931835)

Goal achieved

Remaining resources: 948
Current location: aircraft
Coordinates: (4.0 5.0)

Sensors have located engine

Engine housing opened

Engine repaired

The first two tasks have now been accomplished and the operating system now moves to the less important house-keeping tasks. These last two tasks have been more extensively combined by the planner in order to minimize movements. The vehicle takes inventory at the work-bench, then moves to the sensor area to take inventory, and then moves to the supply room to gather necessary supplies for both the work-bench area and the sensor area. Thus, the vehicle only needs to go to the supply-room one time.

The actual movements between locations have been edited out.

Move Robot
Start Location: aircraft
Destination: work-bench

**** Time Step: 115 ****

New coordinates are (2.000796326710733 2.000000317068165)

Goal achieved

Remaining resources: 935
Current location: work-bench
Coordinates: (2.0 2.0)

Work-bench inventory recalled from Blackboard

Work-bench inventory accomplished

Move Robot

Start Location: work-bench
Destination: sensors

.
.
.

**** Time Step: 127 ****

New coordinates are (5.0 3.0)

Goal achieved

Remaining resources: 925
Current location: sensors
Coordinates: (5.0 3.0)

Sensor inventory recalled from Blackboard

Sensor inventory accomplished

Move Robot

Start Location: sensors
Destination: supply-room

.
.
.

**** Time Step: 137 ****

New coordinates are (4.000796326710733 1.000000317068165)

Goal achieved

Remaining resources: 917
Current location: supply-room
Coordinates: (4.0 1.0)

Sensors have located sensor-supplies

Sensor inventory recalled from Blackboard

Cart loaded with sensor-supplies

Sensors have located work-bench-supplies

Work-bench inventory recalled from Blackboard

Cart loaded with work-bench-supplies

Move Robot

Start Location: supply-room
Destination: work-bench

.
.
.

**** Time Step: 147 ****

New coordinates are (2.000796326710733 1.999999682931835)

Goal achieved

Remaining resources: 909
Current location: work-bench
Coordinates: (2.0 2.0)

Work-bench inventory recalled from Blackboard

work-bench-supplies obtained from cart

work-bench-supplies placed on work-bench

Move Robot

Start Location: work-bench
Destination: sensors

.

.

**** Time Step: 158 ****

New coordinates are (5.0 3.0)

Goal achieved

Remaining resources: 900
Current location: sensors
Coordinates: (5.0 3.0)

Sensor inventory recalled from Blackboard

sensor-supplies obtained from cart

sensors repaired with sensor-supplies

The vehicle has accomplished all of its tasks and now
returns to its base location to await further instructions.

Move Robot

Start Location: sensors
Destination: base

.

**** Time Step: 183 ****

New coordinates are (0.0007963267107332947 3.170681653480445E-07)

Goal achieved

Remaining resources: 882

Current location: base

Coordinates: (0.0 0.0)

t

-> (exit)

VII. Summary, Conclusions, and Recommendations

Summary and Conclusions

One of the goals of this thesis effort was to simulate the processing environment in an autonomous vehicle in order to study the interactions between critical components. The critical components, in this case, are the operating system, mission planner, and route planner. Particular emphasis was given to the operating system and the mission planner, since their functions are crucial to the overall mission of the autonomous vehicle. The route planning process, while equally important in function, has been studied extensively elsewhere. The goal here was to implement a software architecture that could make effective use of all the vehicle's resources.

The architecture adopted for the processing environment was a blackboard control architecture. The implementation here demonstrates the flexibility of such an architecture when applied to the domain of autonomous vehicle planning. The mission planner transforms the task of planning for a specific problem into the task of specifying an appropriate control flow for the operating system. The architecture explicitly represents domain and control problems and integrates the problem solving process into a single basic control loop. Thus, the architecture adapts its problem solving knowledge and its basic control loop to specific problem solving situations (7:283-284).

By partitioning the operating system into knowledge specialists that each perform a basic function, a great deal of flexibility can be achieved. Adding a new capability to the vehicle simply means adding a new knowledge specialist. When to use the added knowledge specialist is then specified by the mission planner. If an additional arm is added to the vehicle, the mission planner's rule base need only be modified to include this additional knowledge. The operating system is not told when to use it, or how to use it except when the mission planner decides it should be used. However, the operating system does require the capability to define its own control flow should the one specified by the mission planner fail. As was shown in Chapter VI, occasions may arise where the world model changes in such a manner as to cause a task failure. Hence, the operating system must be able to specify a temporary control flow in order to recover from such an occurrence. It recovers by detecting new goals and formulating these goals into a structure the mission planner can understand.

The job of goal detection illustrates an important component of the architecture. The operating system formulates the goals into policies for the mission planner, which then uses its meta-knowledge to plan for these policies. The mission planner's control flow is specified by the planner's meta-planning knowledge; therefore, flexibility, similar to the operating system's, is gained by declaratively encoding the planner's control flow in the meta-rules. If a new capability is added to the planner, when and how to use that capability is specified in its

rule base and not buried in a procedure.

Meta-knowledge can also be used to understand the actions of other autonomous vehicles. By "observing" the actions of other vehicles in the environment, the operating system can formulate those actions into goals. The mission planner can then use its meta-knowledge to try and deduce what policies the other vehicle is following. In this manner, the autonomous vehicle can determine if it can share actions with another vehicle in order to accomplish a task. The ability to understand actions, as well as plan for them, are crucial to a truly autonomous vehicle. The architecture of the planning system provides this capability by separating knowledge about how to plan from knowledge about the particular problems.

Finally, the plan projection phase carried out by the operating system enables flaws in the basic plan to be detected. By simulating the components of the plan, the operating system can detect any inconsistencies in the world model, or uncover events that would interrupt the actual plan execution. Thus, more efficient plans can be constructed if errors are detected during the projection phase rather than during the execution phase. Likewise, the plan projection phase can be used during the understanding process to compare components of a hypothesized plan to the actual plan of another vehicle. Inconsistencies would be passed back to the mission planner who would then try to repair the hypothesized plan. One mechanism, therefore, can be used both for understanding and planning due to the explicit nature of the mission planner's knowledge base.

Recommendations

In order to fully demonstrate the capability of this architecture, the route planning function needs to be upgraded. The very simple route planner implemented was sufficient to demonstrate the goals of this thesis; however, the rigid structure of the route planner limited the strategies the vehicle could use to accomplish its tasks. For example, the use of the mission planner to avoid dynamic objects could not be demonstrated since this would have required a route planner with the ability to plan routes from arbitrary starting points in the environment. An improvement in the route planning function would allow a greater variety of tasks to be accomplished and would further demonstrate the flexibility of the operating system.

With the upgrade of the route planner, the sensor actors should be enhanced. Particular emphasis should be given to the sonar actors and their job of detecting objects in the environment. This would imply the need to interface the algorithms used by the sonars with the world model. Thus, the operating system could use the sonar output to identify objects in the environment, thereby enhancing its plan projection capability. Furthermore, this new capability could be used to simulate actually carrying out the plans in the environment.

Although the current implementation is limited to planning for four tasks, the addition of new tasks should not require a great expansion of the knowledge base. The planning rules are encoded in a very general format and should be applicable to a variety of tasks. The meta-planning rules, however, should be

upgraded to demonstrate more of Wilensky's meta-planning concepts. This would require that the appropriate functions be programmed and then their specific use be encoded in the meta-rules.

The mission-planner makes wide use of global variables during the planning phase that may hamper future upgrade of the planning system. The global variables should be eliminated or moved into the blackboard. Likewise, the planner's knowledge base should be moved into the blackboard where it can be shared more easily with other components of the operating system.

The planning system does not have an explicit "understand" mode. With the current implementation, this capability would not be difficult to add. The required sensory ability would have to be canned, of course, but the ability to understand would create a more powerful planning system.

The planner's ability to judge plan values should be refined. The current implementation assigns arbitrary numbers to each task, and the planner merely considers the tasks with a higher number as being more important. No attempt was made to define an approach to assigning plan values, and the result was that the plans were unjustifiably ordered. By defining an approach, the planner could better reason about the tasks it is planning for.

Finally, the meta-planning knowledge base could be broken down into levels. This would allow the control flow in the planner to be more fully specified. Currently, the control flow is specified at a very abstract level with only the main, or driving, functions specified. Therefore, if a meta-plan calls

for the plans to be combined, only the top-level routine is specified in the meta-rule. The top-level routine then specifies how to combine plans. By creating another meta-plan level, control flow in particular functions could be specified, thereby enhancing the planner's ability to explain its actions. For example, the combine function will only combine low-value plans; therefore, if no low-value plans exist, the planner could detect this and use this as a means to explain why it couldn't combine plans. This would be easy to detect since the planner has specified the control flow in the combine function using a meta-plan; therefore, the failure could be pinpointed as occurring in one of the meta-plans.

Appendix: Test Run Transcript

The following is a complete test run of the operating and planning system designed in this thesis. The tasks the planner is to accomplish are:

```
(setq robot-task
  ((action (actor (vehicle (name (robot-1))))
    (refuel aircraft)
    (with nozzle-1))
   (action (actor (vehicle (name (robot-1))))
    (repair engine)
    (location aircraft)
    (with engine-tools))
   (action (actor (vehicle (name (robot-1))))
    (maintain work-bench)
    (with work-bench-supplies))
   (action (actor (vehicle (name (robot-1))))
    (maintain sensors)
    (with sensor-supplies))))
```

->

----- ROSS -----

VERSION: Wed Jan 19 11:26:26 1983

nil

-> (load 'load.r)

t

-> (load 'oper.r)

route defined

t

-> (load 'load.1)

link defined

terminal defined

mvar defined

molecule defined

t
-> (ask scheduler go 300)

*** ROBOT SIMULATION ***
Current Coordinates: (0.0 0.0)
Current Location: base
Current Resources: 100

Ready to plan

Looking for a meta-plan for Policy:

(policy (planner robot)
 (objective (plan (efficiently robot-task))))

Possible explanation assuming

(accomplish (clear-globals all)
 (process-cds robot-task)
 (combine-plans *task-plans*)
 (process-cds *return-action*)
 (refine nil))

A meta-plan for this policy is:

((clear-globals all) (process-cds robot-task)
 (combine-plans *task-plans*)
 (process-cds *return-action*)
 (refine nil))

Planning

(action (actor (vehicle (name (robot-1))))
 (refuel aircraft)
 (with nozzle-1))

Event not predicted

Possible explanation assuming

(refuel-plan (planner (vehicle (name (robot-1))))
 (object aircraft)
 (with nozzle-1))

Event not predicted

Possible explanation assuming

(goal (planner (vehicle (name (robot-1))))
 (objective
 (prox (actor (vehicle (name (robot-1))))))

(location aircraft)
(with nozzle-1)))

Event not predicted

Possible explanation assuming

(move-plan (planner (vehicle (name (robot-1))))
 (from (prox (location nil)))
 (to (prox (location aircraft)))
 (with nozzle-1))

Event not predicted

No usable inferences from

(move-plan (planner (vehicle (name (robot-1))))
 (from (prox (location nil)))
 (to (prox (location aircraft)))
 (with nozzle-1))

No usable inferences from

(goal (planner (vehicle (name (robot-1))))
 (objective
 (prox (actor (vehicle (name (robot-1))))
 (location aircraft)
 (with nozzle-1))))

No usable inferences from

(refuel-plan (planner (vehicle (name (robot-1))))
 (object aircraft)
 (with nozzle-1))

No usable inferences from

(action (actor (vehicle (name (robot-1))))
 (refuel aircraft)
 (with nozzle-1))

No inference chain found--

Seeking Meta-Plan for:

(action (actor (vehicle (name (robot-1))))
 (refuel aircraft)
 (with nozzle-1))

Possible explanation assuming

(accomplish

```
(alternative-scenario
  (action (actor (vehicle (name (robot-1))))
    (refuel aircraft)
    (with nozzle-1))))
```

Trying Alternative Scenario

Possible explanation assuming

```
(accomplish
  (justify-alternative
    (action (actor (vehicle (name (robot-1))))
      (refuel aircraft)
      (with nozzle-2))))
```

Planning

```
(action (actor (vehicle (name (robot-1))))
  (refuel aircraft)
  (with nozzle-2))
```

Event not predicted

Possible explanation assuming

```
(refuel-plan (planner (vehicle (name (robot-1))))
  (object aircraft)
  (with nozzle-2))
```

Event not predicted

Possible explanation assuming

```
(goal (planner (vehicle (name (robot-1))))
  (objective
    (prox (actor (vehicle (name (robot-1))))
      (location aircraft)
      (with nozzle-2))))
```

Event not predicted

Possible explanation assuming

```
(move-plan (planner (vehicle (name (robot-1))))
  (from (prox (location filling-station)))
  (to (prox (location aircraft)))
  (with nozzle-2))
```

Event not predicted

Possible explanation assuming

```
(goal (planner (vehicle (name (robot-1))))  
      (objective  
        (grasp (object nozzle-2)  
                (location filling-station)))))
```

Event not predicted

Possible explanation assuming

```
(grasp-plan (planner (vehicle (name (robot-1))))  
            (object nozzle-2)  
            (location filling-station))
```

Event not predicted

Possible explanation assuming

```
(goal (planner (vehicle (name (robot-1))))  
      (objective  
        (prox (actor (vehicle (name (robot-1))))  
              (location filling-station)  
              (with nil)))))
```

Event not predicted

Possible explanation assuming

```
(move-plan (planner (vehicle (name (robot-1))))  
           (from (prox (location current-location)))  
           (to (prox (location filling-station)))  
           (with nil))
```

Event predicted from

```
(action (actor (vehicle (name (robot-1))))  
        (refuel aircraft)  
        (with nozzle-2))
```

Adding inference chain to data base

Planning

```
(action (actor (vehicle (name (robot-1))))  
        (repair engine)  
        (location aircraft)  
        (with engine-tools))
```

Event not predicted

Possible explanation assuming

```
(repair-engine-plan (planner (vehicle (name (robot-1))))
```


(object engine)
(location aircraft)
(with engine-tools))

Event not predicted

Possible explanation assuming

(goal (planner (vehicle (name (robot-1))))
 (objective
 (prox (actor (vehicle (name (robot-1))))
 (location aircraft)
 (with engine-tools))))

Event not predicted

Possible explanation assuming

(move-plan (planner (vehicle (name (robot-1))))
 (from (prox (location work-bench)))
 (to (prox (location aircraft)))
 (with engine-tools))

Event not predicted

Possible explanation assuming

(goal (planner (vehicle (name (robot-1))))
 (objective
 (grasp (object engine-tools)
 (location work-bench))))

Event not predicted

Possible explanation assuming

(grasp-plan (planner (vehicle (name (robot-1))))
 (object engine-tools)
 (location work-bench))

Event not predicted

Possible explanation assuming

(goal (planner (vehicle (name (robot-1))))
 (objective
 (prox (actor (vehicle (name (robot-1))))
 (location work-bench)
 (with nil))))

Event not predicted

Possible explanation assuming

(move-plan (planner (vehicle (name (robot-1))))

```
(from (prox (location current-location)))  
(to (prox (location work-bench)))  
(with nil))
```

Event predicted from

```
(action (actor (vehicle (name (robot-1))))  
  (repair engine)  
  (location aircraft)  
  (with engine-tools))
```

Adding inference chain to data base

Planning

```
(action (actor (vehicle (name (robot-1))))  
  (maintain work-bench)  
  (with work-bench-supplies))
```

Event not predicted

Possible explanation assuming

```
(replenish-work-bench-plan  
  (planner (vehicle (name (robot-1))))  
  (maintain work-bench)  
  (with work-bench-supplies))
```

Event not predicted

Possible explanation assuming

```
(goal (planner (vehicle (name (robot-1))))  
  (objective  
    (prox (actor (vehicle (name (robot-1))))  
      (location work-bench)  
      (with work-bench-supplies))))
```

Event not predicted

Possible explanation assuming

```
(move-plan (planner (vehicle (name (robot-1))))  
  (from (prox (location supply-room)))  
  (to (prox (location work-bench)))  
  (with work-bench-supplies))
```

Event not predicted

Possible explanation assuming

```
(goal (planner (vehicle (name (robot-1))))  
  (objective
```

```
(load-cart (actor (vehicle (name (robot-1))))
            (location supply-room)
            (with work-bench-supplies)
            (using (inventory
                    (location work-bench)))))
```

Event not predicted

Possible explanation assuming

```
(get-supply-plan (planner (vehicle (name (robot-1))))
                  (location supply-room)
                  (supplies work-bench-supplies)
                  (inventory work-bench))
```

Event not predicted

Possible explanation assuming

```
(goal (planner (vehicle (name (robot-1))))
      (objective
        (prox (actor (vehicle (name (robot-1))))
              (from work-bench)
              (to supply-room)
              (with (knowledge
                    (inventory work-bench-supplies)))))
```

Event not predicted

Possible explanation assuming

```
(move-plan (planner (vehicle (name (robot-1))))
            (from (prox (location work-bench)))
            (to (prox (location supply-room)))
            (with (knowledge (inventory work-bench-supplies))))
```

Event not predicted

Possible explanation assuming

```
(goal (planner (vehicle (name (robot-1))))
      (objective
        (do (actor (vehicle (name (robot-1))))
            (inventory work-bench))))
```

Event not predicted

Possible explanation assuming

```
(inventory-plan (planner (vehicle (name (robot-1))))
                 (location work-bench))
```

Event not predicted

Possible explanation assuming

```
(goal (planner (vehicle (name (robot-1))))
      (objective
        (prox (actor (vehicle (name (robot-1))))
              (location work-bench)
              (with nil))))
```

Event not predicted

Possible explanation assuming

```
(move-plan (planner (vehicle (name (robot-1))))
            (from (prox (location current-location))
                  (to (prox (location work-bench))
                      (with nil)))
```

Event predicted from

```
(action (actor (vehicle (name (robot-1))))
        (maintain work-bench)
        (with work-bench-supplies))
```

Adding inference chain to data base

Planning

```
(action (actor (vehicle (name (robot-1))))
        (maintain sensors)
        (with sensor-supplies))
```

Event not predicted

Possible explanation assuming

```
(sensor-repair-plan (planner (vehicle (name (robot-1))))
                    (maintain sensors)
                    (with sensor-supplies))
```

Event not predicted

Possible explanation assuming

```
(goal (planner (vehicle (name (robot-1))))
      (objective
        (prox (actor (vehicle (name (robot-1))))
              (location sensors)
              (with sensor-supplies))))
```

Event not predicted

Possible explanation assuming

```
(move-plan (planner (vehicle (name (robot-1))))
```

```
(from (prox (location supply-room)))  
(to (prox (location sensors)))  
(with sensor-supplies))
```

Event not predicted

Possible explanation assuming

```
(goal (planner (vehicle (name (robot-1))))  
  (objective  
    (load-cart (actor (vehicle (name (robot-1))))  
      (location supply-room)  
      (with sensor-supplies)  
      (using (inventory (location sensors)))))))
```

Event not predicted

Possible explanation assuming

```
(get-supply-plan (planner (vehicle (name (robot-1))))  
  (location supply-room)  
  (supplies sensor-supplies)  
  (inventory sensors))
```

Event not predicted

Possible explanation assuming

```
(goal (planner (vehicle (name (robot-1))))  
  (objective  
    (prox (actor (vehicle (name (robot-1))))  
      (from sensors)  
      (to supply-room)  
      (with (knowledge  
        (inventory sensor-supplies)))))))
```

Event not predicted

Possible explanation assuming

```
(move-plan (planner (vehicle (name (robot-1))))  
  (from (prox (location sensors)))  
  (to (prox (location supply-room)))  
  (with (knowledge (inventory sensor-supplies))))
```

Event not predicted

Possible explanation assuming

```
(goal (planner (vehicle (name (robot-1))))  
  (objective  
    (do (actor (vehicle (name (robot-1))))  
      (inventory sensors))))
```

Event not predicted

Possible explanation assuming

```
(inventory-plan (planner (vehicle (name (robot-1))))  
                (location sensors))
```

Event not predicted

Possible explanation assuming

```
(goal (planner (vehicle (name (robot-1))))  
      (objective  
        (prox (actor (vehicle (name (robot-1))))  
              (location sensors)  
              (with nil))))
```

Event not predicted

Possible explanation assuming

```
(move-plan (planner (vehicle (name (robot-1))))  
           (from (prox (location current-location)))  
           (to (prox (location sensors)))  
           (with nil))
```

Event predicted from

```
(action (actor (vehicle (name (robot-1))))  
        (maintain sensors)  
        (with sensor-supplies))
```

Adding inference chain to data base

Combining Plans

Sorting Plans by value

Planning

```
(action (actor (vehicle (name (robot-1))))  
        (objective (return (location base))))
```

Event not predicted

Possible explanation assuming

```
(goal (planner (vehicle (name (robot-1))))  
      (objective  
        (prox (actor (vehicle (name (robot-1))))  
              (location base)  
              (with nil))))
```

Event not predicted

Possible explanation assuming

```
(move-plan (planner (vehicle (name (robot-1))))
  (from (prox (location current-location)))
  (to (prox (location base)))
  (with nil))
```

Event predicted from

```
(action (actor (vehicle (name (robot-1))))
  (objective (return (location base))))
```

Adding inference chain to data base

The database contains:

```
((move-plan (planner (vehicle (name (robot-1))))
  (from (prox (location current-location)))
  (to (prox (location filling-station)))
  (with nil)) (a00005)
  m00007)
(grasp-plan (planner (vehicle (name (robot-1))))
  (object nozzle-2)
  (location filling-station)) (a00005)
  m00008)
(move-plan (planner (vehicle (name (robot-1))))
  (from (prox (location filling-station)))
  (to (prox (location aircraft)))
  (with nozzle-2)) (a00005)
  m00009)
(refuel-plan (planner (vehicle (name (robot-1))))
  (object aircraft)
  (with nozzle-2)) (a00005)
  nil)
(move-plan (planner (vehicle (name (robot-1))))
  (from (prox (location current-location)))
  (to (prox (location work-bench)))
  (with nil)) (a00010)
  m00012)
(grasp-plan (planner (vehicle (name (robot-1))))
  (object engine-tools)
  (location work-bench)) (a00010)
  m00013)
(move-plan (planner (vehicle (name (robot-1))))
  (from (prox (location work-bench)))
  (to (prox (location aircraft)))
  (with engine-tools)) (a00010)
  m00014)
(repair-engine-plan (planner (vehicle (name (robot-1))))
  (object engine))
```

```

        (location aircraft)
        (with engine-tools)) (a00010)
                                nil)
((move-plan (planner (vehicle (name (robot-1))))
  (from (prox (location current-location)))
  (to (prox (location work-bench)))
  (with nil)) (a00015)
  m00017)
((inventory-plan (planner (vehicle (name (robot-1))))
  (location work-bench)) (a00015)
  m00018)
((move-plan (planner (vehicle (name (robot-1))))
  (from (prox (location current-location)))
  (to (prox (location sensors)))
  (with nil)) (a00022)
  m00024)
((inventory-plan (planner (vehicle (name (robot-1))))
  (location sensors)) (a00022)
  m00025)
((move-plan (planner (vehicle (name (robot-1))))
  (from (prox (location sensors)))
  (to (prox (location supply-room)))
  (with (knowledge (inventory sensor-supplies))
    (also (knowledge (inventory work-bench-supplies)))))
(a00015 a00022)
m00026)
((get-supply-plan (planner (vehicle (name (robot-1))))
  (location supply-room)
  (supplies sensor-supplies)
  (inventory sensors)) (a00022)
  m00027)
((get-supply-plan (planner (vehicle (name (robot-1))))
  (location supply-room)
  (supplies work-bench-supplies)
  (inventory work-bench)) (a00015)
  m00020)
((move-plan (planner (vehicle (name (robot-1))))
  (from (prox (location supply-room)))
  (to (prox (location work-bench)))
  (with work-bench-supplies)) (a00015)
  m00021)
((replenish-work-bench-plan
  (planner (vehicle (name (robot-1))))
  (maintain work-bench)
  (with work-bench-supplies)) (a00015)
  nil)
((move-plan (planner (vehicle (name (robot-1))))
  (to (prox (location sensors)))
  (with sensor-supplies)
  (from (prox (location work-bench)))) (a00022)
  m00028)
((sensor-repair-plan (planner (vehicle (name (robot-1))))
  (maintain sensors)
  (with sensor-supplies)) (a00022)
  nil)

```



```
((move-plan (planner (vehicle (name (robot-1))))
  (from (prox (location current-location)))
  (to (prox (location base)))
  (with nil)) (a00029)
  nil))
```

Refining the Plans

Refining

```
(move-plan (planner (vehicle (name (robot-1))))
  (from (prox (location current-location)))
  (to (prox (location filling-station)))
  (with nil))
```

Refining

```
(grasp-plan (planner (vehicle (name (robot-1))))
  (object nozzle-2)
  (location filling-station))
```

Refining

```
(move-plan (planner (vehicle (name (robot-1))))
  (from (prox (location filling-station)))
  (to (prox (location aircraft)))
  (with nozzle-2))
```

Refining

```
(refuel-plan (planner (vehicle (name (robot-1))))
  (object aircraft)
  (with nozzle-2))
```

Refining

```
(move-plan (planner (vehicle (name (robot-1))))
  (from (prox (location current-location)))
  (to (prox (location work-bench)))
  (with nil))
```

Refining

```
(grasp-plan (planner (vehicle (name (robot-1))))
  (object engine-tools)
  (location work-bench))
```

Refining

```
(move-plan (planner (vehicle (name (robot-1))))
  (from (prox (location work-bench)))
  (to (prox (location aircraft)))
  (with engine-tools))
```

Refining

```
(repair-engine-plan (planner (vehicle (name (robot-1))))  
                    (object engine)  
                    (location aircraft)  
                    (with engine-tools))
```

Refining

```
(move-plan (planner (vehicle (name (robot-1))))  
           (from (prox (location current-location)))  
           (to (prox (location work-bench)))  
           (with nil))
```

Refining

```
(inventory-plan (planner (vehicle (name (robot-1))))  
                (location work-bench))
```

Refining

```
(move-plan (planner (vehicle (name (robot-1))))  
           (from (prox (location current-location)))  
           (to (prox (location sensors)))  
           (with nil))
```

Refining

```
(inventory-plan (planner (vehicle (name (robot-1))))  
                (location sensors))
```

Refining

```
(move-plan (planner (vehicle (name (robot-1))))  
           (from (prox (location sensors)))  
           (to (prox (location supply-room)))  
           (with (knowledge (inventory sensor-supplies)))  
           (also (knowledge (inventory work-bench-supplies)))))
```

Refining

```
(get-supply-plan (planner (vehicle (name (robot-1))))  
                 (location supply-room)  
                 (supplies sensor-supplies)  
                 (inventory sensors))
```

Refining

```
(get-supply-plan (planner (vehicle (name (robot-1))))  
                 (location supply-room)  
                 (supplies work-bench-supplies)  
                 (inventory work-bench))
```

Refining

```
(move-plan (planner (vehicle (name (robot-1))))
```

```

(from (prox (location supply-room)))
(to (prox (location work-bench)))
(with work-bench-supplies))

```

Refining

```

(replenish-work-bench-plan
  (planner (vehicle (name (robot-1)))))
  (maintain work-bench)
  (with work-bench-supplies))

```

Refining

```

(move-plan (planner (vehicle (name (robot-1)))))
  (to (prox (location sensors)))
  (with sensor-supplies)
  (from (prox (location work-bench))))

```

Refining

```

(sensor-repair-plan (planner (vehicle (name (robot-1)))))
  (maintain sensors)
  (with sensor-supplies))

```

Refining

```

(move-plan (planner (vehicle (name (robot-1)))))
  (from (prox (location current-location)))
  (to (prox (location base)))
  (with nil))

```

The database now contains:

```

(((executive (tell (route-planner))
  (start current-location)
  (goal filling-station)) (a00005)
  m00032)
((executive (tell (scheduler))
  (move route)) (a00005)
  nil)
((executive (tell (sensor))
  (locate nozzle-2)) (a00005)
  m00034)
((executive (tell (arm))
  (open hand)) (a00005)
  m00035)
((executive (tell (arm))
  (move nozzle-2)) (a00005)
  m00036)
((executive (tell (arm))
  (close hand)) (a00005)
  m00037)
((executive (tell (arm))

```

```

                (retract arm)) (a00005)
                        nil)
((executive (tell (route-planner))
              (start filling-station)
              (goal aircraft)) (a00005)
                        m00039)
((executive (tell (scheduler))
              (move route)) (a00005)
                        nil)
((executive (tell (sensor))
              (locate aircraft)) (a00005)
                        m00041)
((executive (tell (sensor))
              (locate filler-cap)) (a00005)
                        m00042)
((executive (tell (arm))
              (open hand)) (a00005)
                        m00043)
((executive (tell (arm))
              (move filler-cap)) (a00005)
                        m00044)
((executive (tell (arm))
              (close hand)) (a00005)
                        m00045)
((executive (tell (arm))
              (rotate arm)) (a00005)
                        m00046)
((executive (tell (arm))
              (retract arm)) (a00005)
                        m00047)
((executive (tell (arm))
              (insert nozzle-2)) (a00005)
                        nil)
((executive (tell (route-planner))
              (start current-location)
              (goal work-bench)) (a00010)
                        m00049)
((executive (tell (scheduler))
              (move route)) (a00010)
                        nil)
((executive (tell (sensor))
              (locate engine-tools)) (a00010)
                        m00051)
((executive (tell (arm))
              (open hand)) (a00010)
                        m00052)
((executive (tell (arm))
              (move engine-tools)) (a00010)
                        m00053)
((executive (tell (arm))
              (close hand)) (a00010)
                        m00054)
((executive (tell (arm))
              (retract arm)) (a00010)
                        nil)

```

```

((executive (tell (route-planner))
              (start work-bench)
              (goal aircraft)) (a00010)
              m00056)
((executive (tell (scheduler))
              (move route)) (a00010)
              nil)
((executive (tell (sensor))
              (locate engine)) (a00010)
              m00058)
((executive (tell (arm))
              (open engine)) (a00010)
              m00059)
((executive (tell (arm))
              (repair engine)) (a00010)
              nil)
((executive (tell (route-planner))
              (start current-location)
              (goal work-bench)) (a00015)
              m00061)
((executive (tell (scheduler))
              (move route)) (a00015)
              nil)
((executive (tell (blackboard))
              (recall (inventory (location work-bench))))
(a00015)
m00063)
((executive (tell (scheduler))
              (do (inventory (location work-bench)))) (a00015)
              nil)
((executive (tell (route-planner))
              (start current-location)
              (goal sensors)) (a00022)
              m00065)
((executive (tell (scheduler))
              (move route)) (a00022)
              nil)
((executive (tell (blackboard))
              (recall (inventory (location sensors)))) (a00022)
              m00067)
((executive (tell (scheduler))
              (do (inventory (location sensors)))) (a00022)
              nil)
((executive (tell (route-planner))
              (start sensors)
              (goal supply-room)) (a00015 a00022)
              m00069)
((executive (tell (scheduler))
              (move route)) (a00015 a00022)
              nil)
((executive (tell (sensor))
              (locate sensor-supplies)) (a00022)
              m00071)
((executive (tell (blackboard))
              (recall (inventory (location sensors)))) (a00022)

```

m00072)

```
((executive (tell (scheduler))
              (load cart)
              (with sensor-supplies)) (a00022)
              nil)

((executive (tell (sensor))
              (locate work-bench-supplies)) (a00015)
              m00074)

((executive (tell (blackboard))
              (recall (inventory (location work-bench))))
(a00015)
m00075)

((executive (tell (scheduler))
              (load cart)
              (with work-bench-supplies)) (a00015)
              nil)

((executive (tell (route-planner))
              (start supply-room)
              (goal work-bench)) (a00015)
              m00077)

((executive (tell (scheduler))
              (move route)) (a00015)
              nil)

((executive (tell (blackboard))
              (recall (inventory (location work-bench))))
(a00015)
m00079)

((executive (tell (scheduler))
              (get work-bench-supplies)
              (from cart)) (a00015)
              m00080)

((executive (tell (scheduler))
              (put work-bench-supplies)
              (on work-bench)) (a00015)
              nil)

((executive (tell (route-planner))
              (start work-bench)
              (goal sensors)) (a00022)
              m00082)

((executive (tell (scheduler))
              (move route)) (a00022)
              nil)

((executive (tell (blackboard))
              (recall (inventory (location sensors)))) (a00022)
              m00084)

((executive (tell (scheduler))
              (get sensor-supplies)
              (from cart)) (a00022)
              m00085)

((executive (tell (scheduler))
              (repair sensors)
              (with sensor-supplies)) (a00022)
              nil)

((executive (tell (route-planner))
              (start current-location)
```

```
(goal base)) (a00029)
                m00087)
((executive (tell (scheduler))
            (move route)) (a00029)
            nil))
```

Move Robot
Start Location: base
Destination: filling-station

Encoder initializes memory
Sonar initializes memory
Gyro initializes memory

**** Time step: 1 ****

Confirm initialization

Initialization Confirmed
Ready For Motion Simulation

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 2 ****

New coordinates are (0.5 0.0)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 3 ****

New coordinates are (1.0 0.0)

New route step

New heading has been computed

Turn left 45 degrees
Reorient Robot 45 degrees

**** Time step: 4 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 5 ****

New coordinates are (1.314835507758704 0.3884309501755995)

Course deviation -- calling errorhandler

New heading has been computed

Turn right -9.227130622162847 degrees
Reorient Robot -9.227130622162847 degrees

**** Time step: 6 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 7 ****

New coordinates are (1.687854361240208 0.7213828009698912)

New route step

New heading has been computed

Turn left 3.227130622162847 degrees
Reorient Robot 3.227130622162847 degrees

**** Time step: 8 ****

Robot moves forward: 0.5 meters

Encoder updates memory

Sonar updates memory
Gyro updates memory

**** Time Step: 9 ****

New coordinates are (2.334723082637353 1.371430286796524)

Course deviation -- calling errorhandler

New heading has been computed

Turn right -4.603079295651878 degrees
Reorient Robot -4.603079295651878 degrees

**** Time step: 10 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 11 ****

New coordinates are (2.673335159021275 1.739318668954712)

New route step

New heading has been computed

Turn right -47.39692070434812 degrees
Reorient Robot -47.39692070434812 degrees

**** Time step: 12 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 13 ****

New coordinates are (3.5 2.0)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 14 ****

New coordinates are (4.0 2.0)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 15 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 16 ****

New coordinates are (4.5 2.0)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 17 ****

New coordinates are (5.0 2.0)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 18 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 19 ****

New coordinates are (5.5 2.0)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 20 ****

New coordinates are (6.0 2.0)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 21 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 22 ****

New coordinates are (6.5 2.0)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 23 ****

New coordinates are (7.0 2.0)

New route step

New heading has been computed

Turn left 90.0 degrees
Reorient Robot 90.0 degrees

**** Time step: 24 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 25 ****

New coordinates are (7.000398163355367 2.499999841465917)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 26 ****

New coordinates are (7.000796326710733 2.999999682931835)

Goal achieved

Remaining resources: 78
Current location: filling-station
Coordinates: (7.0 3.0)

 Robot Fuel Critically Low

Looking for a meta-plan for Policy:

```
(policy (planner robot)
  (objective (plan (maximize *low-robot-fuel-task*)))))
```

Possible explanation assuming

```
(accomplish (save-top-level-tasks nil)
  (clear-globals some)
  (process-cds *low-robot-fuel-task*)
  (process-cds *return-suspend-action*)
  (sort *task-plans*)
  (refine nil)
  (restore-top-level-tasks nil))
```

A meta-plan for this policy is:

```
((save-top-level-tasks nil) (clear-globals some)
  (process-cds
    *low-robot-fuel-task*)
  (process-cds
    *return-suspend-action*)
  (sort *task-plans*)
  (refine nil)
  (restore-top-level-tasks nil))
```

Planning

```
(action (actor (vehicle (name (robot-1))))
  (correct (state (low robot-fuel))))
```

Event not predicted

Possible explanation assuming

```
(refuel-robot-plan (planner (vehicle (name (robot-1))))
  (correct (state (low robot-fuel)))
  (with suspended-state))
```

Event not predicted

Possible explanation assuming

```
(goal (planner (vehicle (name (robot-1))))
  (objective
    (prox (actor (vehicle (name (robot-1))))
      (location robot-fuel)
      (with suspended-state))))
```

Event not predicted

Possible explanation assuming

```
(move-plan (planner (vehicle (name (robot-1))))  
  (from (prox (location suspend-location)))  
  (to (prox (location robot-fuel)))  
  (with suspended-state))
```

Event not predicted

Possible explanation assuming

```
(goal (planner (vehicle (name (robot-1))))  
  (objective (suspend (task current-task))))
```

Event not predicted

Possible explanation assuming

```
(suspend-state-plan (planner (vehicle (name (robot-1)))))
```

Event predicted from

```
(action (actor (vehicle (name (robot-1))))  
  (correct (state (low robot-fuel))))
```

Adding inference chain to data base

Planning

```
(action (actor (vehicle (name (robot-1))))  
  (objective (return (location suspend-location))))
```

Event not predicted

Possible explanation assuming

```
(goal (planner (vehicle (name (robot-1))))  
  (objective  
    (prox (actor (vehicle (name (robot-1))))  
      (location suspend-location)  
      (with nil))))
```

Event not predicted

Possible explanation assuming

```
(move-plan (planner (vehicle (name (robot-1))))  
  (from (prox (location current-location)))  
  (to (prox (location suspend-location)))  
  (with nil))
```

Event predicted from

```
(action (actor (vehicle (name (robot-1))))  
        (objective (return (location suspend-location)))))
```

Adding inference chain to data base

The database contains:

```
((suspend-state-plan (planner (vehicle (name (robot-1)))))  
(a00095)  
m00097)  
((move-plan (planner (vehicle (name (robot-1)))))  
  (from (prox (location suspend-location)))  
  (to (prox (location robot-fuel)))  
  (with suspended-state)) (a00095)  
                             m00098)  
((refuel-robot-plan (planner (vehicle (name (robot-1)))))  
  (correct (state (low robot-fuel)))  
  (with suspended-state)) (a00095)  
                             nil)  
((move-plan (planner (vehicle (name (robot-1)))))  
  (from (prox (location current-location)))  
  (to (prox (location suspend-location)))  
  (with nil)) (a00099)  
               nil))
```

Sorting Plans by value

Refining the Plans

Refining

```
(suspend-state-plan (planner (vehicle (name (robot-1)))))
```

Refining

```
(move-plan (planner (vehicle (name (robot-1)))))  
  (from (prox (location suspend-location)))  
  (to (prox (location robot-fuel)))  
  (with suspended-state))
```

Refining

```
(refuel-robot-plan (planner (vehicle (name (robot-1)))))  
  (correct (state (low robot-fuel)))  
  (with suspended-state))
```

Refining

```
(move-plan (planner (vehicle (name (robot-1)))))  
  (from (prox (location current-location)))  
  (to (prox (location suspend-location)))
```

(with nil))

The database now contains:

```
((executive (tell (blackboard))
              (save state)) (a00095)
              nil)
((executive (tell (route-planner))
              (start suspend-location)
              (goal robot-fuel)) (a00095)
              m00103)
((executive (tell (scheduler))
              (move route)) (a00095)
              nil)
((executive (tell (sensor))
              (locate refuel-socket)) (a00095)
              m00105)
((executive (tell (arm))
              (move refuel-socket)) (a00095)
              m00106)
((executive (tell (arm))
              (insert robot-finger)) (a00095)
              m00107)
((executive (tell (scheduler))
              (robot refueled)) (a00095)
              nil)
((executive (tell (route-planner))
              (start current-location)
              (goal suspend-location)) (a00099)
              m00109)
((executive (tell (scheduler))
              (move route)) (a00099)
              nil))
```

Move Robot

Start Location: filling-station

Destination: robot-fuel

Encoder initializes memory

Sonar initializes memory

Gyro initializes memory

**** Time step: 27 ****

Confirm initialization

Heading error in initialization

Calling Error Handler

New heading has been computed

Turn right -180.0 degrees
Reorient Robot -180.0 degrees

Re-initialize memory

Encoder initializes memory
Sonar initializes memory
Gyro initializes memory

**** Time step: 28 ****

Confirm initialization

Initialization Confirmed
Ready For Motion Simulation

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 29 ****

New coordinates are (7.000398163355367 2.500000158534083)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 30 ****

New coordinates are (7.000796326710733 2.000000317068165)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 31 ****

New coordinates are (7.0011944900661 1.500000475602248)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 32 ****

New coordinates are (7.001592653421467 1.000000634136331)

Goal achieved

Remaining resources: 73
Current location: robot-fuel
Coordinates: (7.0 1.0)

Sensors have located refuel-socket

Arm moved to refuel-socket

robot-finger has been inserted

Robot Refueled
Current resources: 1000

Move Robot
Start Location: robot-fuel
Destination: filling-station

Encoder initializes memory
Sonar initializes memory
Gyro initializes memory

**** Time step: 33 ****

Confirm initialization

Heading error in initialization
Calling Error Handler

New heading has been computed

Turn left 180.0 degrees

Reorient Robot 180.0 degrees

Re-initialize memory

Encoder initializes memory

Sonar initializes memory

Gyro initializes memory

**** Time step: 34 ****

Confirm initialization

Initialization Confirmed

Ready For Motion Simulation

Robot moves forward: 0.5 meters

Encoder updates memory

Sonar updates memory

Gyro updates memory

**** Time Step: 35 ****

New coordinates are (7.000398163355367 1.499999841465917)

Robot moves forward: 0.5 meters

Encoder updates memory

Sonar updates memory

Gyro updates memory

**** Time Step: 36 ****

New coordinates are (7.000796326710733 1.999999682931835)

Robot moves forward: 0.5 meters

Encoder updates memory

Sonar updates memory

Gyro updates memory

**** Time Step: 37 ****

New coordinates are (7.0011944900661 2.499999524397752)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 38 ****

New coordinates are (7.001592653421467 2.999999365863669)

Goal achieved

Remaining resources: 995
Current location: filling-station
Coordinates: (7.0 3.0)

Sensors cannot Locate: nozzle-2

Task Failure Detected

Failed Function: (a00005)

Modifying Policy

Looking for a meta-plan for Policy:

```
(policy (planner robot)
  (objective (plan (failed a00005))))
```

Possible explanation assuming

```
(accomplish (abandon a00005)
  (move-to-top-level nil)
  (clear-globals some)
  (extern-plan-fail a00005)
  (sort *task-plans*)
  (push-new-plan nil)
  (refine nil))
```

A meta-plan for this policy is:

```
((abandon a00005) (move-to-top-level nil)
  (clear-globals some)
  (extern-plan-fail a00005)
  (sort *task-plans*)
  (push-new-plan nil)
  (refine nil))
```

Abandoning Plan a00005

Seeking Meta-Plan for:

```
(action (actor (vehicle (name (robot-1))))  
        (refuel aircraft)  
        (with nozzle-2))
```

Possible explanation assuming

```
(accomplish  
  (alternative-scenario  
    (action (actor (vehicle (name (robot-1))))  
            (refuel aircraft)  
            (with nozzle-2))))
```

Trying Alternative Scenario

Possible explanation assuming

```
(accomplish  
  (justify-alternative  
    (action (actor (vehicle (name (robot-1))))  
            (refuel aircraft)  
            (with nozzle-3))))
```

Planning

```
(action (actor (vehicle (name (robot-1))))  
        (refuel aircraft)  
        (with nozzle-3))
```

Event not predicted

Possible explanation assuming

```
(refuel-plan (planner (vehicle (name (robot-1))))  
             (object aircraft)  
             (with nozzle-3))
```

Event not predicted

Possible explanation assuming

```
(goal (planner (vehicle (name (robot-1))))  
      (objective  
        (prox (actor (vehicle (name (robot-1))))
```

(location aircraft)
(with nozzle-3)))

Event not predicted

Possible explanation assuming

(move-plan (planner (vehicle (name (robot-1))))
 (from (prox (location filling-station-2)))
 (to (prox (location aircraft)))
 (with nozzle-3))

Event not predicted

Possible explanation assuming

(goal (planner (vehicle (name (robot-1))))
 (objective
 (grasp (object nozzle-3)
 (location filling-station-2))))

Event not predicted

Possible explanation assuming

(grasp-plan (planner (vehicle (name (robot-1))))
 (object nozzle-3)
 (location filling-station-2))

Event not predicted

Possible explanation assuming

(goal (planner (vehicle (name (robot-1))))
 (objective
 (prox (actor (vehicle (name (robot-1))))
 (location filling-station-2)
 (with nil))))

Event not predicted

Possible explanation assuming

(move-plan (planner (vehicle (name (robot-1))))
 (from (prox (location current-location)))
 (to (prox (location filling-station-2)))
 (with nil))

Event predicted from

(action (actor (vehicle (name (robot-1))))
 (refuel aircraft)
 (with nozzle-3))

Adding inference chain to data base

Sorting Plans by value

Refining the Plans

Refining

```
(move-plan (planner (vehicle (name (robot-1))))  
  (from (prox (location current-location)))  
  (to (prox (location filling-station-2)))  
  (with nil))
```

Refining

```
(grasp-plan (planner (vehicle (name (robot-1))))  
  (object nozzle-3)  
  (location filling-station-2))
```

Refining

```
(move-plan (planner (vehicle (name (robot-1))))  
  (from (prox (location filling-station-2)))  
  (to (prox (location aircraft)))  
  (with nozzle-3))
```

Refining

```
(refuel-plan (planner (vehicle (name (robot-1))))  
  (object aircraft)  
  (with nozzle-3))
```

Refining

```
(move-plan (planner (vehicle (name (robot-1))))  
  (from (prox (location current-location)))  
  (to (prox (location work-bench)))  
  (with nil))
```

Refining

```
(grasp-plan (planner (vehicle (name (robot-1))))  
  (object engine-tools)  
  (location work-bench))
```

Refining

```
(move-plan (planner (vehicle (name (robot-1))))  
  (from (prox (location work-bench)))  
  (to (prox (location aircraft)))  
  (with engine-tools))
```

Refining

```
(repair-engine-plan (planner (vehicle (name (robot-1))))  
                    (object engine)  
                    (location aircraft)  
                    (with engine-tools))
```

Refining

```
(move-plan (planner (vehicle (name (robot-1))))  
           (from (prox (location current-location)))  
           (to (prox (location work-bench)))  
           (with nil))
```

Refining

```
(inventory-plan (planner (vehicle (name (robot-1))))  
                (location work-bench))
```

Refining

```
(move-plan (planner (vehicle (name (robot-1))))  
           (from (prox (location current-location)))  
           (to (prox (location sensors)))  
           (with nil))
```

Refining

```
(inventory-plan (planner (vehicle (name (robot-1))))  
                (location sensors))
```

Refining

```
(move-plan (planner (vehicle (name (robot-1))))  
           (from (prox (location sensors)))  
           (to (prox (location supply-room)))  
           (with (knowledge (inventory sensor-supplies)))  
           (also (knowledge (inventory work-bench-supplies)))))
```

Refining

```
(get-supply-plan (planner (vehicle (name (robot-1))))  
                 (location supply-room)  
                 (supplies sensor-supplies)  
                 (inventory sensors))
```

Refining

```
(get-supply-plan (planner (vehicle (name (robot-1))))  
                 (location supply-room)  
                 (supplies work-bench-supplies)  
                 (inventory work-bench))
```

Refining

```
(move-plan (planner (vehicle (name (robot-1))))
```



```

      (from (prox (location supply-room)))
      (to (prox (location work-bench)))
      (with work-bench-supplies))

```

Refining

```

(replenish-work-bench-plan
  (planner (vehicle (name (robot-1)))))
  (maintain work-bench)
  (with work-bench-supplies))

```

Refining

```

(move-plan (planner (vehicle (name (robot-1)))))
  (to (prox (location sensors)))
  (with sensor-supplies)
  (from (prox (location work-bench))))

```

Refining

```

(sensor-repair-plan (planner (vehicle (name (robot-1)))))
  (maintain sensors)
  (with sensor-supplies))

```

Refining

```

(move-plan (planner (vehicle (name (robot-1)))))
  (from (prox (location current-location)))
  (to (prox (location base)))
  (with nil))

```

The database now contains:

```

(((executive (tell (route-planner))
  (start current-location)
  (goal filling-station-2)) (a00125)
  m00131)
 ((executive (tell (scheduler))
  (move route)) (a00125)
  nil)
 ((executive (tell (sensor))
  (locate nozzle-3)) (a00125)
  m00133)
 ((executive (tell (arm))
  (open hand)) (a00125)
  m00134)
 ((executive (tell (arm))
  (move nozzle-3)) (a00125)
  m00135)
 ((executive (tell (arm))
  (close hand)) (a00125)
  m00136)
 ((executive (tell (arm))

```

```

        (retract arm)) (a00125)
                        nil)
((executive (tell (route-planner))
              (start filling-station-2)
              (goal aircraft)) (a00125)
                        m00138)
((executive (tell (scheduler))
              (move route)) (a00125)
                        nil)
((executive (tell (sensor))
              (locate aircraft)) (a00125)
                        m00140)
((executive (tell (sensor))
              (locate filler-cap)) (a00125)
                        m00141)
((executive (tell (arm))
              (open hand)) (a00125)
                        m00142)
((executive (tell (arm))
              (move filler-cap)) (a00125)
                        m00143)
((executive (tell (arm))
              (close hand)) (a00125)
                        m00144)
((executive (tell (arm))
              (rotate arm)) (a00125)
                        m00145)
((executive (tell (arm))
              (retract arm)) (a00125)
                        m00146)
((executive (tell (arm))
              (insert nozzle-3)) (a00125)
                        nil)
((executive (tell (route-planner))
              (start current-location)
              (goal work-bench)) (a00010)
                        m00148)
((executive (tell (scheduler))
              (move route)) (a00010)
                        nil)
((executive (tell (sensor))
              (locate engine-tools)) (a00010)
                        m00150)
((executive (tell (arm))
              (open hand)) (a00010)
                        m00151)
((executive (tell (arm))
              (move engine-tools)) (a00010)
                        m00152)
((executive (tell (arm))
              (close hand)) (a00010)
                        m00153)
((executive (tell (arm))
              (retract arm)) (a00010)
                        nil)

```

```

((executive (tell (route-planner))
              (start work-bench)
              (goal aircraft)) (a00010)
              m00155)
((executive (tell (scheduler))
              (move route)) (a00010)
              nil)
((executive (tell (sensor))
              (locate engine)) (a00010)
              m00157)
((executive (tell (arm))
              (open engine)) (a00010)
              m00158)
((executive (tell (arm))
              (repair engine)) (a00010)
              nil)
((executive (tell (route-planner))
              (start current-location)
              (goal work-bench)) (a00015)
              m00160)
((executive (tell (scheduler))
              (move route)) (a00015)
              nil)
((executive (tell (blackboard))
              (recall (inventory (location work-bench))))
(a00015)
m00162)
((executive (tell (scheduler))
              (do (inventory (location work-bench)))) (a00015)
              nil)
((executive (tell (route-planner))
              (start current-location)
              (goal sensors)) (a00022)
              m00164)
((executive (tell (scheduler))
              (move route)) (a00022)
              nil)
((executive (tell (blackboard))
              (recall (inventory (location sensors)))) (a00022)
              m00166)
((executive (tell (scheduler))
              (do (inventory (location sensors)))) (a00022)
              nil)
((executive (tell (route-planner))
              (start sensors)
              (goal supply-room)) (a00015 a00022)
              m00168)
((executive (tell (scheduler))
              (move route)) (a00015 a00022)
              nil)
((executive (tell (sensor))
              (locate sensor-supplies)) (a00022)
              m00170)
((executive (tell (blackboard))
              (recall (inventory (location sensors)))) (a00022)

```

m00171)

```
((executive (tell (scheduler))
              (load cart)
              (with sensor-supplies)) (a00022)
              nil)

((executive (tell (sensor))
              (locate work-bench-supplies)) (a00015)
              m00173)

((executive (tell (blackboard))
              (recall (inventory (location work-bench))))
(a00015)
m00174)
((executive (tell (scheduler))
              (load cart)
              (with work-bench-supplies)) (a00015)
              nil)

((executive (tell (route-planner))
              (start supply-room)
              (goal work-bench)) (a00015)
              m00176)

((executive (tell (scheduler))
              (move route)) (a00015)
              nil)

((executive (tell (blackboard))
              (recall (inventory (location work-bench))))
(a00015)
m00178)
((executive (tell (scheduler))
              (get work-bench-supplies)
              (from cart)) (a00015)
              m00179)

((executive (tell (scheduler))
              (put work-bench-supplies)
              (on work-bench)) (a00015)
              nil)

((executive (tell (route-planner))
              (start work-bench)
              (goal sensors)) (a00022)
              m00181)

((executive (tell (scheduler))
              (move route)) (a00022)
              nil)

((executive (tell (blackboard))
              (recall (inventory (location sensors)))) (a00022)
              m00183)

((executive (tell (scheduler))
              (get sensor-supplies)
              (from cart)) (a00022)
              m00184)

((executive (tell (scheduler))
              (repair sensors)
              (with sensor-supplies)) (a00022)
              nil)

((executive (tell (route-planner))
              (start current-location)
```

```
          (goal base)) (a00029)
                        m00186)
((executive (tell (scheduler))
            (move route)) (a00029)
            nil))
```

Move Robot

Start Location: filling-station
Destination: filling-station-2

Encoder initializes memory
Sonar initializes memory
Gyro initializes memory

**** Time step: 39 ****

Confirm initialization

Heading error in initialization
Calling Error Handler

New heading has been computed

Turn right -90.0 degrees
Reorient Robot -90.0 degrees

Re-initialize memory

Encoder initializes memory
Sonar initializes memory
Gyro initializes memory

**** Time step: 40 ****

Confirm initialization

Initialization Confirmed
Ready For Motion Simulation

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 41 ****

New coordinates are (7.5 3.0)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 42 ****

New coordinates are (8.0 3.0)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 43 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 44 ****

New coordinates are (8.5 3.0)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 45 ****

New coordinates are (9.0 3.0)

Goal achieved

Remaining resources: 990
Current location: filling-station-2
Coordinates: (9.0 3.0)

Sensors have located nozzle-3

Hand opened

Arm moved to nozzle-3

Hand closed

Arm retracted

Move Robot

Start Location: filling-station-2
Destination: aircraft

Encoder initializes memory
Sonar initializes memory
Gyro initializes memory

**** Time step: 46 ****

Confirm initialization

Heading error in initialization
Calling Error Handler

New heading has been computed

Turn left 90.0 degrees
Reorient Robot 90.0 degrees

Re-initialize memory

Encoder initializes memory
Sonar initializes memory
Gyro initializes memory

**** Time step: 47 ****

Confirm initialization

Initialization Confirmed
Ready For Motion Simulation

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 48 ****

New coordinates are (9.000398163355367 3.499999841465917)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 49 ****

New coordinates are (9.000796326710733 3.999999682931835)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 50 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 51 ****

New coordinates are (9.000398163355367 4.499999841465917)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 52 ****

New coordinates are (9.000796326710733 4.999999682931835)

New route step

New heading has been computed

Turn left 90.0 degrees
Reorient Robot 90.0 degrees

**** Time step: 53 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 54 ****

New coordinates are (8.50000063413623 5.000796326458243)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 55 ****

New coordinates are (8.00000126827246 5.001592652916487)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 56 ****

Robot moves forward: 0.5 meters

Encoder updates memory

Sonar updates memory

Gyro updates memory

**** Time Step: 57 ****

New coordinates are (7.50000063413623 5.000796326458243)

Robot moves forward: 0.5 meters

Encoder updates memory

Sonar updates memory

Gyro updates memory

**** Time Step: 58 ****

New coordinates are (7.00000126827246 5.001592652916487)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 59 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 60 ****

New coordinates are (6.50000063413623 5.000796326458243)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 61 ****

New coordinates are (6.00000126827246 5.001592652916487)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 62 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 63 ****

New coordinates are (5.50000063413623 5.000796326458243)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 64 ****

New coordinates are (5.00000126827246 5.001592652916487)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 65 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 66 ****

New coordinates are (4.50000063413623 5.000796326458243)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 67 ****

New coordinates are (4.00000126827246 5.001592652916487)

Goal achieved

Remaining resources: 974
Current location: aircraft
Coordinates: (4.0 5.0)

Sensors have located aircraft

Sensors have located filler-cap

Hand opened

Arm moved to filler-cap

Hand closed

Arm rotated

Arm retracted

nozzle-3 has been inserted

Move Robot

Start Location: aircraft

Destination: work-bench

Encoder initializes memory

Sonar initializes memory

Gyro initializes memory

**** Time step: 68 ****

Confirm initialization

Heading error in initialization

Calling Error Handler

New heading has been computed

Turn right -270.0 degrees

Reorient Robot -270.0 degrees

Re-initialize memory

Encoder initializes memory

Sonar initializes memory

Gyro initializes memory

**** Time step: 69 ****

Confirm initialization

Initialization Confirmed

Ready For Motion Simulation

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 70 ****

New coordinates are (4.000398163355367 4.500000158534083)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 71 ****

New coordinates are (4.000796326710733 4.000000317068165)

New route step

New heading has been computed

Turn left 270.0 degrees
Reorient Robot 270.0 degrees

**** Time step: 72 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 73 ****

New coordinates are (3.50000063413623 4.000796326458243)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 74 ****

New coordinates are (3.00000126827246 4.001592652916487)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 75 ****

Robot moves forward: 0.5 meters

Encoder updates memory

Sonar updates memory

Gyro updates memory

**** Time Step: 76 ****

New coordinates are (2.50000063413623 4.000796326458243)

Robot moves forward: 0.5 meters

Encoder updates memory

Sonar updates memory

Gyro updates memory

**** Time Step: 77 ****

New coordinates are (2.00000126827246 4.001592652916487)

New route step

New heading has been computed

Turn right -270.0 degrees

Reorient Robot -270.0 degrees

**** Time step: 78 ****

Robot moves forward: 0.5 meters

Encoder updates memory

Sonar updates memory

Gyro updates memory

**** Time Step: 79 ****

New coordinates are (2.000398163355367 3.500000158534083)

Robot moves forward: 0.5 meters

Encoder updates memory

Sonar updates memory

Gyro updates memory

**** Time Step: 80 ****

New coordinates are (2.000796326710733 3.000000317068165)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 81 ****

Robot moves forward: 0.5 meters

Encoder updates memory

Sonar updates memory

Gyro updates memory

**** Time Step: 82 ****

New coordinates are (2.000398163355367 2.500000158534083)

Robot moves forward: 0.5 meters

Encoder updates memory

Sonar updates memory

Gyro updates memory

**** Time Step: 83 ****

New coordinates are (2.000796326710733 2.000000317068165)

Goal achieved

Remaining resources: 961
Current location: work-bench
Coordinates: (2.0 2.0)

Sensors have located engine-tools

Hand opened

Arm moved to engine-tools

Hand closed

Arm retracted

Move Robot

Start Location: work-bench
Destination: aircraft

Encoder initializes memory
Sonar initializes memory
Gyro initializes memory

**** Time step: 84 ****

Confirm initialization

Heading error in initialization
Calling Error Handler

New heading has been computed

Turn left 180.0 degrees
Reorient Robot 180.0 degrees

Re-initialize memory

Encoder initializes memory
Sonar initializes memory
Gyro initializes memory

**** Time step: 85 ****

Confirm initialization

Initialization Confirmed
Ready For Motion Simulation

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 86 ****

New coordinates are (2.000398163355367 2.499999841465917)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 87 ****

New coordinates are (2.000796326710733 2.999999682931835)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 88 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 89 ****

New coordinates are (2.000398163355367 3.499999841465917)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 90 ****

New coordinates are (2.000796326710733 3.999999682931835)

New route step

New heading has been computed

Turn right -90.0 degrees
Reorient Robot -90.0 degrees

**** Time step: 91 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 92 ****

New coordinates are (2.5 4.0)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 93 ****

New coordinates are (3.0 4.0)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 94 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 95 ****

New coordinates are (3.5 4.0)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 96 ****

New coordinates are (4.0 4.0)

New route step

New heading has been computed

Turn left 90.0 degrees
Reorient Robot 90.0 degrees

**** Time step: 97 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 98 ****

New coordinates are (4.000398163355367 4.499999841465917)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 99 ****

New coordinates are (4.000796326710733 4.999999682931835)

Goal achieved

Remaining resources: 948
Current location: aircraft
Coordinates: (4.0 5.0)

Sensors have located engine

Engine housing opened

Engine repaired

Move Robot

Start Location: aircraft
Destination: work-bench

Encoder initializes memory
Sonar initializes memory
Gyro initializes memory

**** Time step: 100 ****

Confirm initialization

Heading error in initialization
Calling Error Handler

New heading has been computed

Turn right -180.0 degrees
Reorient Robot -180.0 degrees

Re-initialize memory

Encoder initializes memory
Sonar initializes memory
Gyro initializes memory

**** Time step: 101 ****

Confirm initialization

Initialization Confirmed
Ready For Motion Simulation

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 102 ****

New coordinates are (4.000398163355367 4.500000158534083)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 103 ****

New coordinates are (4.000796326710733 4.000000317068165)

New route step

New heading has been computed

Turn left 270.0 degrees
Reorient Robot 270.0 degrees

**** Time step: 104 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 105 ****

New coordinates are (3.50000063413623 4.000796326458243)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 106 ****

New coordinates are (3.00000126827246 4.001592652916487)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 107 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 108 ****

New coordinates are (2.50000063413623 4.000796326458243)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 109 ****

New coordinates are (2.00000126827246 4.001592652916487)

New route step

New heading has been computed

Turn right -270.0 degrees
Reorient Robot -270.0 degrees

**** Time step: 110 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 111 ****

New coordinates are (2.000398163355367 3.500000158534083)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 112 ****

New coordinates are (2.000796326710733 3.000000317068165)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 113 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 114 ****

New coordinates are (2.000398163355367 2.500000158534083)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 115 ****

New coordinates are (2.000796326710733 2.000000317068165)

Goal achieved

Remaining resources: 935
Current location: work-bench
Coordinates: (2.0 2.0)

Work-bench inventory recalled from Blackboard

Work-bench inventory accomplished

Move Robot

Start Location: work-bench
Destination: sensors

Encoder initializes memory
Sonar initializes memory
Gyro initializes memory

**** Time step: 116 ****

Confirm initialization

Heading error in initialization
Calling Error Handler

New heading has been computed

Turn left 180.0 degrees
Reorient Robot 180.0 degrees

Re-initialize memory

Encoder initializes memory
Sonar initializes memory
Gyro initializes memory

**** Time step: 117 ****

Confirm initialization

Initialization Confirmed
Ready For Motion Simulation

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 118 ****

New coordinates are (2.000398163355367 2.499999841465917)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 119 ****

New coordinates are (2.000796326710733 2.999999682931835)

New route step

New heading has been computed

Turn right -90.0 degrees
Reorient Robot -90.0 degrees

**** Time step: 120 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 121 ****

New coordinates are (2.5 3.0)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 122 ****

New coordinates are (3.0 3.0)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 123 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 124 ****

New coordinates are (3.5 3.0)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 125 ****

New coordinates are (4.0 3.0)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 126 ****

New coordinates are (4.5 3.0)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 127 ****

New coordinates are (5.0 3.0)

Goal achieved

Remaining resources: 925
Current location: sensors
Coordinates: (5.0 2.0)

Sensor inventory recalled from Blackboard

Sensor inventory accomplished

Move Robot
Start Location: sensors
Destination: supply-room

Encoder initializes memory
Sonar initializes memory
Gyro initializes memory

**** Time step: 128 ****

Confirm initialization

Heading error in initialization
Calling Error Handler

New heading has been computed

Turn left 180.0 degrees
Reorient Robot 180.0 degrees

AD-A163 956

AUTONOMOUS VEHICLE MISSION PLANNING USING AI
(ARTIFICIAL INTELLIGENCE) TE. (U) AIR FORCE INST OF
TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI..

3/3

UNCLASSIFIED

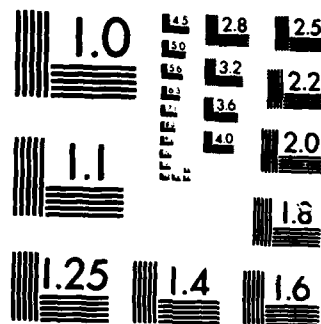
S E STOCKBRIDGE DEC 85 AFIT/GE/ENG/85D-45 F/G 6/4

NL

END

FILED

BTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Re-initialize memory

Encoder initializes memory
Sonar initializes memory
Gyro initializes memory

**** Time step: 129 ****

Confirm initialization

Initialization Confirmed
Ready For Motion Simulation

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 130 ****

New coordinates are (4.50000063413623 3.000796326458243)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 131 ****

New coordinates are (4.00000126827246 3.001592652916487)

New route step

New heading has been computed

Turn right -270.0 degrees
Reorient Robot -270.0 degrees

**** Time step: 132 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory

Gyro updates memory

**** Time Step: 133 ****

New coordinates are (4.000398163355367 2.500000158534083)

Robot moves forward: 0.5 meters

Encoder updates memory

Sonar updates memory

Gyro updates memory

**** Time Step: 134 ****

New coordinates are (4.000796326710733 2.000000317068165)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 135 ****

Robot moves forward: 0.5 meters

Encoder updates memory

Sonar updates memory

Gyro updates memory

**** Time Step: 136 ****

New coordinates are (4.000398163355367 1.500000158534083)

Robot moves forward: 0.5 meters

Encoder updates memory

Sonar updates memory

Gyro updates memory

**** Time Step: 137 ****

New coordinates are (4.000796326710733 1.000000317068165)

Goal achieved

Remaining resources: 917
Current location: supply-room
Coordinates: (4.0 1.0)

Sensors have located sensor-supplies

Sensor inventory recalled from Blackboard

Cart loaded with sensor-supplies

Sensors have located work-bench-supplies

Work-bench inventory recalled from Blackboard

Cart loaded with work-bench-supplies

Move Robot

Start Location: supply-room
Destination: work-bench

Encoder initializes memory
Sonar initializes memory
Gyro initializes memory

**** Time step: 138 ****

Confirm initialization

Heading error in initialization
Calling Error Handler

New heading has been computed

Turn left 270.0 degrees
Reorient Robot 270.0 degrees

Re-initialize memory

Encoder initializes memory
Sonar initializes memory
Gyro initializes memory

**** Time step: 139 ****

Confirm initialization

Initialization Confirmed
Ready For Motion Simulation

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 140 ****

New coordinates are (3.50000063413620 1.000796326458243)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 141 ****

New coordinates are (3.00000126827246 1.001592652916487)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 142 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 143 ****

New coordinates are (2.50000063413623 1.000796326458243)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 144 ****

New coordinates are (2.00000126827246 1.001592652916487)

New route step

New heading has been computed

Turn right -90.0 degrees
Reorient Robot -90.0 degrees

**** Time step: 145 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 146 ****

New coordinates are (2.000398163355367 1.499999841465917)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 147 ****

New coordinates are (2.000796326710733 1.999999682931835)

Goal achieved

Remaining resources: 909
Current location: work-bench

Coordinates: (2.0 2.0)

Work-bench inventory recalled from Blackboard

work-bench-supplies obtained from cart

work-bench-supplies placed on work-bench

Move Robot

Start Location: work-bench

Destination: sensors

Encoder initializes memory

Sonar initializes memory

Gyro initializes memory

**** Time step: 148 ****

Confirm initialization

Initialization Confirmed

Ready For Motion Simulation

Robot moves forward: 0.5 meters

Encoder updates memory

Sonar updates memory

Gyro updates memory

**** Time Step: 149 ****

New coordinates are (2.000398163355367 2.499999841465917)

Robot moves forward: 0.5 meters

Encoder updates memory

Sonar updates memory

Gyro updates memory

**** Time Step: 150 ****

New coordinates are (2.000796326710733 2.999999682931835)

New route step

New heading has been computed

Turn right -90.0 degrees
Reorient Robot -90.0 degrees

**** Time step: 151 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 152 ****

New coordinates are (2.5 3.0)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 153 ****

New coordinates are (3.0 3.0)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 154 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 155 ****

New coordinates are (3.5 3.0)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 156 ****

New coordinates are (4.0 3.0)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 157 ****

New coordinates are (4.5 3.0)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 158 ****

New coordinates are (5.0 3.0)

Goal achieved

Remaining resources: 900
Current location: sensors
Coordinates: (5.0 3.0)

Sensor inventory recalled from Blackboard

sensor-supplies obtained from cart

sensors repaired with sensor-supplies

Move Robot

Start Location: sensors

Destination: base

Encoder initializes memory

Sonar initializes memory

Gyro initializes memory

**** Time step: 159 ****

Confirm initialization

Heading error in initialization

Calling Error Handler

New heading has been computed

Turn left 180.0 degrees

Reorient Robot 180.0 degrees

Re-initialize memory

Encoder initializes memory

Sonar initializes memory

Gyro initializes memory

**** Time step: 160 ****

Confirm initialization

Initialization Confirmed

Ready For Motion Simulation

Robot moves forward: 0.5 meters

Encoder updates memory

Sonar updates memory

Gyro updates memory

**** Time Step: 161 ****

New coordinates are (4.50000063413623 3.000796326458243)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 162 ****

New coordinates are (4.00000126827246 3.001592652916487)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 163 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 164 ****

New coordinates are (3.50000063413623 3.000796326458243)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 165 ****

New coordinates are (3.00000126827246 3.001592652916487)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 166 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 167 ****

New coordinates are (2.50000063413623 3.000796326458243)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 168 ****

New coordinates are (2.00000126827246 3.001592652916487)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 169 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 170 ****

New coordinates are (1.50000063413623 3.000796326458243)

Robot moves forward: 0.5 meters

Encoder updates memory

Sonar updates memory
Gyro updates memory

**** Time Step: 171 ****

New coordinates are (1.00000126827246 3.001592652916487)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 172 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 173 ****

New coordinates are (0.5000006341362302 3.000796326458243)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 174 ****

New coordinates are (1.268272460400177E-06 3.001592652916487)

New route step

New heading has been computed

Turn right -270.0 degrees
Reorient Robot -270.0 degrees

**** Time step: 175 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 176 ****

New coordinates are (0.0003981633553666474 2.500000158534083)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 177 ****

New coordinates are (0.0007963267107332947 2.000000317068165)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 178 ****

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 179 ****

New coordinates are (0.0003981633553666474 1.500000158534083)

Robot moves forward: 0.5 meters

Encoder updates memory
Sonar updates memory
Gyro updates memory

**** Time Step: 180 ****

New coordinates are (0.0007963267107332947 1.000000317068165)

New route step

New heading has been computed

Continue same heading for new segment

**** Time step: 181 ****

Robot moves forward: 0.5 meters

Encoder updates memory

Sonar updates memory

Gyro updates memory

**** Time Step: 182 ****

New coordinates are (0.0003981633553666474 0.5000001585340827)

Robot moves forward: 0.5 meters

Encoder updates memory

Sonar updates memory

Gyro updates memory

**** Time Step: 183 ****

New coordinates are (0.0007963267107332947 3.170681653480445E-07)

Goal achieved

Remaining resources: 882

Current location: base

Coordinates: (0.0 0.0)

t
-> (exit)

Bibliography

1. Deitel, Harvey M. An Introduction to Operating Systems. Reading, MA: Addison-Wesley, 1984.
2. Charniak, Eugene et al. Artificial Intelligence Programming. Hillsdale, NJ: Lawrence Erlbaum Associates, 1980.
3. Clifford, Thomas and Hubert Schneider. Creating a Mobile Autonomous Robot Research System (MARRS). MS thesis GE/ENG/84D-19. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1984.
4. Cohen, Paul R. et al (eds.) "Planning and Problem Solving," Handbook of Artificial Intelligence, 3: 515-522 (1983).
5. Crowley, James L. "Navigation for an Intelligent Mobile Robot," IEEE Journal of Robotics and Automation, RA-1 (1): 31-41 (March 1985).
6. Giralt, Georges et al. "A Multi-Level Planning and Navigation System for a Mobile Robot; A First Approach to Hilare," Proceedings IJAI-79, 1: 335-337 (1979).
7. Hayes-Roth, Barbara. "A Blackboard Architecture for Control," Artificial Intelligence: An International Journal, 26 (3): 251-321 (July 1985).
8. Hayes-Roth, Frederick et al. "Modeling Planning as an Incremental, Opportunistic Process," Proceedings IJAI-79, 2: 375-383 (1979).
9. Keirsey, D. et al. "Autonomous Vehicle Control Using AI Techniques," IEEE Transactions on Software Engineering, SE-11 (9): 986-992 (September 1985).
10. Monaghan, Glen. Navigation for an Autonomous Mobile Robot. MS thesis GE/ENG/84D-47. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1984.
11. Nitzan, David. "Development of Intelligent Robots: Achievements and Issues," IEEE Journal of Robotics and Automation, RA-1 (1): 3-13 (March 1985).

12. Owen, Robert. Environmental Mapping by a HERO-1 Robot Using Sonar and a Laser Barcode Scanner. MS thesis GE/EE/83D-52. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1983.
13. Rich, Elaine. Artificial Intelligence. New York, NY: McGraw-Hill, 1984.
14. Schank, Roger C. and Christopher K. Riesbeck. Inside Computer Understanding: Five Programs Plus Miniatures. Hillsdale, NJ: Lawrence Erlbaum Associates, 1981.
15. Wilensky, Robert. Planning and Understanding. Reading, MA: Addison-Wesley, 1983.
16. -----. "Meta-Planning: Representing and Using Knowledge About Planning in Problem Solving and Natural Language Understanding," Cognitive Science, 5: 197-233 (1981).

VITA

Captain Samuel E. Stockbridge was born on 19 February 1958 in Laredo, Texas. He graduated from high school in Laredo, Texas in 1976 and attended the University of Texas at Austin from which he received the degree of Bachelor of Science in Electrical Engineering in August of 1980. Upon graduation, he received a commission in the USAF through Officer Training School. In March of 1981, he was assigned to the Air Force Weapons Laboratory, Kirtland AFB, New Mexico, where he worked as a Laser Vulnerability Engineer until entering the School of Engineering, Air Force Institute of Technology, in June 1984.

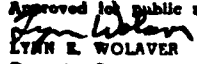
permanent address: 1019 Cortez St.
Laredo, Texas 78040

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

AD-A163 956

REPORT DOCUMENTATION PAGE

1. REPORT SECURITY CLASSIFICATION Unclassified			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GE/ENG/85D-45			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
6a. NAME OF PERFORMING ORGANIZATION School of Engineering		6b. OFFICE SYMBOL (If applicable) AFIT/ENG		7a. NAME OF MONITORING ORGANIZATION	
6c. ADDRESS (City, State and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB, OH 45433				7b. ADDRESS (City, State and ZIP Code)	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AF AFRL		8b. OFFICE SYMBOL (If applicable) BBA		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State and ZIP Code) Aerospace Medical Research Labs Wright-Patterson AFB, OH 45433				10. SOURCE OF FUNDING NOS.	
11. TITLE (Include Security Classification) See BOX 19				PROGRAM ELEMENT NO.	PROJECT NO.
				TASK NO.	WORK UNIT NO.
12. PERSONAL AUTHOR(S) Samuel E. Stockbridge, BSSE, Capt, USAF					
13a. TYPE OF REPORT Thesis		13b. TIME COVERED FROM _____ TO _____		14. DATE OF REPORT (Yr., Mo., Day) 1985 December	
15. PAGE COUNT 210					
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB. GR.	Artificial Intelligence Planning		
09	02		Computerized Simulation		
06	04				
19. ABSTRACT (Continue on reverse if necessary and identify by block number)					
Title: AUTONOMOUS VEHICLE MISSION PLANNING USING AI TECHNIQUES					
Thesis Chairman: Dr. M. Abrisky Professor of Electrical Engineering					
Approved for public release IAW AFR 100-17.  LYNN E. WOLAVER 16 JAN 86 Dean for Research and Professional Development Air Force Institute of Technology (AFIT) Wright-Patterson AFB OH 45433					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐			21. ABSTRACT SECURITY CLASSIFICATION Unclassified		
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr. M. Abrisky			22b. TELEPHONE NUMBER (Include Area Code) 513-255-5276		22c. OFFICE SYMBOL AFIT/ENG

11-515
This study investigates a software architecture for autonomous vehicle control. The autonomous vehicle's planning ability is divided into operating system functions and mission planning system functions. The blackboard control architecture is adopted for the operating system design with implementation using the ROSS programming language.

The planning system incorporates elements of a planner and understander by declaratively encoding meta-knowledge, or knowledge about the planning process. By separating the knowledge about how to plan from the specific domain knowledge, an understander can use this knowledge about how plans are constructed, in combination with the specific domain knowledge, in the understanding process. Likewise, a planner can use this same knowledge in the planning process. Thus, a great deal of flexibility is attained by dividing the knowledge base into meta-rules and domain specific rules.

The planning system constructs an agenda of scripts which directs the control flow in the operating system. The operating system is given the additional duties of goal detector and plan projector and detects any errors in the plan.

The implementation demonstrates the benefits of using meta-planning concepts combined with a blackboard control architecture to provide an autonomous vehicle with a more flexible and powerful planning capability.

END

FILMED

3-86

DTIC