NAVAL POSTGRADUATE SCHOOL
Monterey, California

DISSERTATION

THE RATIONAL BEHAVIOR MODEL:
A MULTI-PARADIGM, TRI-LEVEL SOFTWARE
ARCHITECTURE FOR THE CONTROL OF
AUTONOMOUS VEHICLES

by

Ronald Benton Byrnes, Jr.

March 1993

Dissertation Co-Supervisor: Robert B. McGhee
Dissertation Co-Supervisor: Michael L. Nelson

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL *(if applicable)* CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS *(City, State, and ZIP Code)* Monterey, California 93943-5000	7b. ADDRESS *(City, State, and ZIP Code)* Monterey, California 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL *(if applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)*	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)* THE RATIONAL BEHAVIOR MODEL: A MULTI-PARADIGM, TRI-LEVEL SOFTWARE ARCHITECTURE FOR THE CONTROL OF AUTONOMOUS VEHICLES

12. PERSONAL AUTHOR(S)
Byrnes, Ronald Benton, Jr.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Ph.D. Dissertation	From To	March 1993	315

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

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	autonomous vehicles, autonomous mobile robots, software architectures, automated reasoning, vehicle control software, intelligent control systems

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

There is currently a very strong interest among researchers in the fields of artificial intelligence and robotics in finding more effective means of linking high level symbolic computations relating to mission planning and control for autonomous vehicles to low level vehicle control software. The diversity exhibited by the many processes involved in such control has resulted in a number of proposals for a general *software architecture* intended to provide an efficient yet flexible framework for the organization and interaction of relevant software components. The Rational Behavior Model (RBM) has been developed with these requirements in mind and consists of three levels, called the *Strategic*, the *Tactical*, and the *Execution* levels, respectively. Each level reflects computations supporting the solution to the global control problem based on different abstraction mechanisms. The unique contribution of the RBM architecture is the idea of specifying different programming paradigms to realize each software level. Specifically, RBM uses rule-based programming for the Strategic level, thereby permitting field reconfiguration of missions by a mission specialist without reprogramming at lower levels. The Tactical level realizes vehicle *behaviors* as the methods of software objects programmed in an object-based language such as Ada. These behaviors are initiated by rule satisfaction at the Strategic level, thereby *rationalizing* their interaction. The Execution level is programmed in any imperative language capable of supporting efficient execution of real-time control of the underlying vehicle hardware. The viability of this architecture has been established through computer simulation studies of control of an autonomous submarine, the NPS Autonomous Underwater Vehicle. These experiments have confirmed that the RBM architecture provides important advantages in terms of program conciseness, maintainability, reliability, and modifiability. In addition, by constraining the interfaces between the levels and limiting the accessibility of state variables, the team development of autonomous vehicle control software is significantly enhanced.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Robert B. McGhee	22b. TELEPHONE *(Include Area Code)* (408) 656-2026	22c. OFFICE SYMBOL Code CS/Mz

DD FORM 1473, JUN 86 *Previous editions are obsolete*

Approved for public release; distribution is unlimited

The Rational Behavior Model: a Multi-Paradigm, Tri-level Software Architecture for the Control of Autonomous Vehicles

by

Ronald Benton Byrnes, Jr.
Major, United States Army
B.S., Midwestern State University, 1979
M.S.E.E., Naval Postgraduate School, 1989

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1993

Author: _____
Ronald Benton Byrnes, Jr.

Approved By:

_____
Luqi
Professor of Computer Science

_____
Michael L. Nelson
Professor of Computer Science
Dissertation Co-Supervisor

_____
Shridhar Shukla
Professor of Electrical Engineering

_____
Robert B. McGhee
Professor of Computer Science
Dissertation Co-Supervisor

_____
Anthony J. Healey
Professor of Mechanical Engineering

Approved by: _____
Gary Hughes, Acting Chairman, Department of Computer Science

Approved by: Maurice D. Weir, Assoc. Dean of Instruction
acting for Richard S. Elster, Dean of Instruction

ABSTRACT

There is currently a very strong interest among researchers in the fields of artificial intelligence and robotics in finding more effective means of linking high level symbolic computations relating to mission planning and control for autonomous vehicles to low level vehicle control software. The diversity exhibited by the many processes involved in such control has resulted in a number of proposals for a general *software architecture* intended to provide an efficient yet flexible framework for the organization and interaction of relevant software components. The Rational Behavior Model (RBM) has been developed with these requirements in mind and consists of three levels, called the *Strategic*, the *Tactical*, and the *Execution* levels, respectively. Each level reflects computations supporting the solution to the global control problem based on different abstraction mechanisms. The unique contribution of the RBM architecture is the idea of specifying different programming paradigms to realize each software level. Specifically, RBM uses rule-based programming for the Strategic level, thereby permitting field reconfiguration of missions by a mission specialist without reprogramming at lower levels. The Tactical level realizes vehicle *behaviors* as the methods of software objects programmed in an object-based language such as Ada. These behaviors are initiated by rule satisfaction at the Strategic level, thereby *rationalizing* their interaction. The Execution level is programmed in any imperative language capable of supporting efficient execution of real-time control of the underlying vehicle hardware. The viability of this architecture has been established through computer simulation studies of control of an autonomous submarine, the NPS Autonomous Underwater Vehicle. These experiments have confirmed that the RBM architecture provides important advantages in terms of program conciseness, maintainability, reliability, and modifiability. In addition, by constraining the interfaces between the levels and limiting the accessibility of state variables, the team development of autonomous vehicle control software is significantly enhanced.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

First and foremost, I must acknowledge the unfailing and unconditional support I have received through it all from my wife, Denise, without which this work could never have been completed. Her positive attitude and understanding while dealing with the everyday burdens of child rearing were remarkable as they were essential to my success. I am also indebted to my parents, Dr. Ronald B. and Carolyn G. Byrnes, and my grandparents, Nicholas J. and Mary T. Guillet for instilling in me the value of education and the desire to pursue my goals.

I also wish to express my deepest gratitude to Professor Robert B. McGhee whose support, guidance, and enthusiasm have been a constant inspiration to me. Professor Michael L. Nelson deserves special recognition and thanks for his valuable insights into the Object-Oriented world and his uncanny ability to visualize software solutions from that perspective.

Although not a member of my committee, Professor Se-Hung Kwak invested his time and broad expertise as if he were. His technical support was, of course, essential; however, his constant attention and interest in the frustrating formative stages of this research were equally invaluable.

Finally, I wish to thank the other members of my committee for their support and special insights; the Computer Science Department staff for their outstanding assistance; LCDR Thomas Scholz, Federal German Navy, for his contributions to the experimental studies of this dissertation; and my fellow Ph.D. students who always provided encouragement and comic relief when times got hard.

This research was supported in part by the National Science Foundation under Grant No. BCS-9109989.

I. INTRODUCTION

A. BACKGROUND

There is no question that advances in VLSI circuitry design, processor speed and capabilities, and hardware miniaturization have revolutionized the field of autonomous mobile robots [Ref. 1]. These breakthroughs have led naturally to increasingly sophisticated and complex requirements by the end users of robotic systems, and these same users have grown to expect finished products which satisfy these requirements. It follows that users are identifying applications where fully autonomous robots would be advantageous, either by removing the human element from a dangerous or undesirable work environment or by extending the capability of humans into a new domain. Scenarios in which autonomous robots perform these duties have existed only in the realm of science fiction up to now; however, the technological advances of the past decade have placed us in a position where realization of such robots are not only possible but inevitable [Ref. 2].

However, there are many issues yet to be addressed. It is one thing to design and build the physical robot, and quite another to imbue the robot with the desired degree of intelligence. Experience and expertise in these areas have been accumulating for many decades and is now generally associated with the scientific field of *robotics*. Control of the hardware, including motors, arms, and actuators, is relatively well understood and is usually discussed in the context of feedback control theory [Ref. 3]. This theory is, after all, founded upon the fundamental tenets of mathematics, physics, and engineering. Feedback (or servo) control is ideal when applied to systems, robotic or otherwise, that perform either regulation (disturbance rejection) or command tracking tasks. Examples include position control of manipulator arms associated with industrial robots employed for the assembly of components. In most cases, the "problem" being solved has been subjected to extensive study. Every detail has been identified, analyzed, and programmed from a formal, mathe-

matical perspective. The robot's *work cell* [Ref. 4], or environment, is also rigorously modeled. The goal of these efforts is to minimize uncertainty and risk by expressing the robot's world and its interaction with that world in unambiguous terms.

Research in the field of robotics goes well beyond servo control, however. Concepts from artificial intelligence, computer vision, vehicle navigation, and graph theory are utilized to implement compiled human knowledge as applied to the problems of motion, path planning, and object identification and avoidance [Ref. 5]. Even so, there are many applications where fully *autonomous* robots would be desirable but which cannot be described mathematically, or circumstances where all possible situations cannot be anticipated. The related tasks will, as a result, be complex and wrought with uncertainty. In these circumstances, the static nature of the assumptions pertaining to the environment of the robot will no longer hold, and systems based on these assumptions can no longer be relied upon to perform as expected. Attempting to represent an uncertain world mathematically is very difficult; therefore, a typical solution is to place a human "in the loop" to provide the necessary expertise and to guide the robot through these uncertain situations. It is apparent that truly autonomous solutions to this class of problems will require a different approach. This requirement has given rise to the development of *software architectures* as a means of extending the domain of robotics research into applications involving uncertain, dynamic environments [Ref. 6].

Designing a software architecture for the control of an autonomous vehicle can provide a framework upon which a complete software control system can be built. Despite the relaxing of the requirement to represent the uncertain environment mathematically, the software architecture must still deal with the daunting problem of how best to represent and respond to the world. Another related issue is the management of the software complexity associated with the underlying subsystems, both software and hardware. Experience has shown that large software systems require the integration and coordination of the work of many individuals [Ref. 7]. Without extensive planning, coordination, and organization, resulting programs may be riddled with errors, the identification and correction of which will

prove costly in terms of human and monetary resources [Ref. 8]. Even then, there is no guarantee of perfect software. Undetected errors or interactions between individually correct modules may cause unexpected behavior in the overall system. Just as problematic, the system may not behave as the end user expected, the result of misunderstood or incomplete requirements.

B. SCOPE OF DISSERTATION

This dissertation presents a software architecture for the control of autonomous vehicles operating in a dynamic environment. The problem domain addressed by this work has three major aspects: the control of autonomous vehicles in unstructured environments, the reconfiguration of the missions anticipated by this class of autonomous vehicles, and the reusability of software associated with this control. Each of these issues has, to differing degrees, driven researchers to provide better ways to design and develop control software for autonomous agents. Until now, however, no approach has successfully integrated all three requirements into a single architecture. The Rational Behavior Model (RBM), through the use of appropriate abstractions and the selection of different programming paradigms according to their particular applicability to the problem, has been defined specifically with these problems in mind.

Management of software complexity is the primary purpose of any software architecture. The architecture must define operational and logical domains into which are placed the software systems that accomplish the tasks at hand. Automated reasoning is an essential component of the architecture if the vehicle is to effectively execute a mission while contending with the uncertainty of dynamically changing environments. Other issues, including mission replanning, fault identification and isolation, intervention, and the reaction to unanticipated events which would, under other circumstances, be handled through human intervention, must be taken into account within the architectural framework. This requirement for organization and coordination would almost certainly overwhelm a design using a "traditional" approach, even one involving the functional hierarchies of structured pro-

gramming. Again, the RBM provides a means to overcome this problem through the application of a rich set of abstractions and multiple programming paradigms.

C. ORGANIZATION OF DISSERTATION

Chapter II begins with a survey of previous research work on control software for autonomous vehicles. Following this is a discussion of the requirements associated with autonomous vehicle control and the problem domain relevant to autonomous vehicle operation. Next, a description of the various approaches to control software architectures currently in use is provided. Current, significant autonomous vehicles are then recounted along with a description of the associated software architecture of each. The Naval Postgraduate School Autonomous Underwater Vehicle and its associated simulation facilities are then presented.

Various programming paradigms and languages available to the designer of a software architecture for control of an autonomous vehicle is the focus of Chapter III. The strengths and weaknesses of each, as they relate to this problem, are examined.

In Chapter IV, the topics of computational logic, automated reasoning, and the implementation issues of each are discussed. Two specific approaches to the implementation of reasoning, backward and forward chaining, are described, and graphical representations of each introduced.

The Rational Behavior Model, a tri-level, domain specific software architecture for the overall control of autonomous vehicles is formally defined in Chapter V. Related software control architectures for autonomous vehicle control and the shortcomings of each are presented. Next, the characteristics, requirements, and constraints associated with each level of RBM are described.

Chapter VI describes the results of a simulation study of an instantiation of RBM applied to a real-world scenario. Specific implementation considerations are detailed and the solutions to each are provided. The implementations of RBM developed for this dissertation are explained, followed by discussion, analysis, and evaluation of experimental results.

Finally, Chapter VII summarizes the contributions of the Rational Behavior Model to the field of vehicle control software architectures and provides suggestions for future research. Appendices at the end of the dissertation contain the source code listing for the programs used in the experiments of Chapter VI. A Glossary immediately follows which contains definitions of the many terms and concepts used throughout this dissertation and in common usage in the current literature on autonomous vehicle control software architectures.

II. AUTONOMOUS VEHICLES

This chapter is designed to provide the reader with the background necessary to fully understand the research that follows. To this end, it begins by providing a brief historical survey of the evolution of autonomous mobile robots from a control software perspective. Of particular interest is the evolution of the software organization and logic representation aspects of autonomous vehicle control. Requirements and capabilities to be expected of autonomous vehicles are then described. The next section of this chapter discusses various widely recognized approaches to control software architecture development. Although quite diverse in nature, it is possible to categorize these approaches based on shared characteristics. A brief overview of significant autonomous vehicle research is given. Finally, a section describing the Naval Postgraduate School (NPS) Autonomous Underwater Vehicle (AUV) and associated simulation facilities is presented, followed by a summary.

Due to the relative immaturity and uncoordinated nature of the research in this field, little has been done to standardize terminology describing control software architectures. This has resulted in unnecessary confusion. Therefore, a glossary devoted to the precise definition of common terms and concepts encountered in this area is included at the end of this dissertation. Primarily compiled to support the research described herein, it is offered as a basis for further refinement to the control software community.

A. INTRODUCTION

The class of autonomous vehicles is a specialization of the larger class of mobile robots, which itself is a specialization of the general class of robots. The term *robot*, a term derived from the Czech *robota* ("forced labor") [Ref. 9], refers loosely to a mechanical entity that performs in a seemingly human way. It follows that a mobile robot, besides performing its robotic duties, is capable of moving from one location to another by means of wheels, tracks, legs, thrusters, propellers, etc. Furthermore, the emulation of human behav-

ior implies some degree of *autonomy*, the ability to perform independently of human, or *supervisory*, control. Hence, an autonomous mobile vehicle, or more simply *autonomous vehicle*, is a robot capable of motion and of responding in an intelligent way to a changing environment without human involvement. The distinction between an autonomous mobile robot and an autonomous vehicle is merely one of semantics. The term *vehicle* implies a platform capable of carrying or conveying an object. All autonomous mobile robots carry something, even if the "cargo" consists only of its own sensory and computational devices.

DEFINITION: An *Autonomous Vehicle* is a self-contained mobile robot with the capacity to sense a dynamic and unstructured environment, plan an intelligent response to that input, and act in a way that is compatible with the accomplishment of a mission without human intervention.

All robots are controlled by the basic cycle of *sense*, *decide*, and *act* [Ref. 10]. The sensing portion of the cycle occurs when readings are taken by the robot of its environment. Although much research is currently ongoing in the areas of computer vision, modeling, sensor interpretation, and sensor integration [Ref. 11], the problem of sensing is generally well understood. However, given the speed and resolution of current sensors, the sheer quantity of data can, in fact, overwhelm the robot's decision making process. The action portion of the cycle, manifested in the robot's motion, is likewise well understood [Ref. 12]. The decision phase is the least understood and, hence, has the widest variety of solutions associated with it. This diversity is what differentiates the various approaches to the control software of robots [Ref. 6].

The decision phase of the basic robot control cycle is difficult to realize because the encoding of basic human knowledge is hard. For this reason, a great deal of effort has been invested in vehicles which provide the sensing and action phases of the control cycle but rely on a human operator for the decision-making. This occurs when the robot is teleoperated, either through a radio or cable link[1]. This class of vehicles is called *Remotely Oper-*

1. Obviously, this also occurs when a human driver is integrated into the overall design of the system.

ated Vehicles (ROV) for underwater and land operations and *Remotely Piloted Vehicles* (RPV) for airborne operations. Prominent examples of these classes are the Monterey Bay Aquarium Research Institute's ROV [Ref. 13], the JASON ROV of Woods Hole Oceanographic Institute [Ref. 14], the Navy's Remotely Operated Vehicle for Emplacement and Reconnaissance (ROVER) and Mine Neutralization System (MNS) [Ref. 15], and the Lockheed Aquila [Ref. 15]. It may be argued that because they rely on a human to provide decision making, ROVs are not robots at all but rather a sophisticated extension of the human's sense and reach. On the other hand, the introduction of the new generation of cruise missiles and "smart" bombs blurs boundaries between remotely operated and autonomous vehicles. In any case, ROVs cannot be thought of as autonomous, and information concerning missiles which emulate human reasoning is generally classified; therefore, neither are further discussed in this dissertation.

B. THE DEVELOPMENT OF CONTROL SOFTWARE FOR AUTONOMOUS VEHICLES

The investigation into machine intelligence applied to mobile devices began soon after World War II. However, not until the development of "Shakey" [Ref. 16] in the 1960's was autonomy in a robot demonstrated. Although connected to an external computer through a radio link, and only able to solve simple control problems in a structured environment, Shakey was able to navigate, explore, and learn about its environment without human involvement. The software written to control Shakey was assembled into several levels using the principle of *hierarchy*, a type of software architecture described more fully later in this chapter.

Another significant autonomous vehicle was the Stanford Cart, a TV-equipped mobile robot that also was linked to a remote computer. This system underwent further development, eventually evolving into what became known as the Carnegie-Mellon University (CMU) Rover. During this evolution, the software underwent restructuring as well, result-

ing in a concept of control based on a three level hierarchy communicating via a *blackboard*[2] data structure.[Ref. 17]

The control hierarchy established by Shakey and the CMU rover was probably as much a result of the need for distributed decision making as it was a reaction to software complexity. Limits on computational resources required that different functions, such as perception, navigation, and planning, execute cooperatively on separate processors. This distribution tends to support the top-down approach to many control problems, where high-level planning is followed by navigation and finally path execution [Ref. 18].

Despite the promise of early experiments, research into control of autonomous vehicles began to wane in the 1970's, primarily as a result of the complexity associated with sensory processing. It was not until the advent of the microprocessor that interest was revived. This breakthrough, combined with advances in robotic and sensing technology, ushered in a new era in which autonomous agents were not only endowed with increasing intelligence but could be made to operate in complex, dynamic environments. [Ref. 19]

It had been clear for many years that legged devices demonstrated advantages in improved mobility, isolation, and stability over wheeled and tracked vehicles in difficult terrain or soil conditions [Ref. 20]. Theoretical analysis, supported by experimental evidence, has revealed that six-legged (hexapod) designs are superior to those with two or four legs, both in terms of adaptability and stability. As the number of legs increases, however, so do the attendant control and coordination problems. Work at Ohio State University, which led to the development of the Adaptive Suspension Vehicle (ASV), gained recognition as the first operative multi-legged system to fully solve this problem by means of computer, rather than human, control [Ref. 21]. Hence, although designed to carry a human, the ASV was mechanically autonomous, requiring only high-level steering and velocity commands from the operator. Kwak [Ref. 22] demonstrated the effectiveness of several algorithms for the

2. A blackboard, in this context, is a globally-accessible data store where problem-solving state variables are maintained. Independent software entities acting upon the blackboard produce changes to the problem state, incrementally leading to a solution.

on-line optimization of stepping patterns, and later introduced the concept of rule-based control of the ASV [Ref. 23]. However, despite the successful demonstration of new concepts by the ASV, it proved to be too slow, too bulky, and too expensive to be useful as a production vehicle [Ref. 20].

Work on the control of legged vehicles in unstructured terrain only underscored the challenges faced by researchers when moving their autonomous vehicles from the static, benign environment of the laboratory into the hostile, dynamic world. The nature and extent of a continuously changing environment and the determination of what must be sensed, the selection by the planning agent of an action chosen from an enormous repertoire of possibilities, and the accounting of incomplete data and unforeseen circumstances had to be dealt with by a truly intelligent control system. Such systems must do more than mimic human action. They are required to reason about their world.

One approach to this problem, grounded in traditional artificial intelligence, is the *situational calculus* of McCarthy [Ref. 24]. Within this system, logical terms are used to denote situations (states), events, and *fluents* of the problem domain[3]. In this approach, predicates in the situational calculus are used primarily to describe the context of fluent values in particular states, as well as to specify state transitions associated with an event in the problem domain. The situation calculus also contains the usual logical connectives and quantifiers of first order predicate calculus. Used together, general assertions about the effects of events applied in particular situations can be expressed.

Another logical formalism developed to represent and reason about dynamic domains is *modal logic*. In avoiding the explicit use of terms representing the world state, modal logic ameliorates the need to specify every property of the domain left unaffected by an event. This characteristic of situational calculus is known as the *frame problem* [Ref. 25]. Various

3. The *state* of a system is a collection of attributes uniquely describing the system at a specific instance in time. An *event* is used to record and describe the *behavior* of a system, where a behavior is a sequence of world states. A *fluent* is a function corresponding to a shared property between world states and whose value in a given state is the value of that property in the state.[Ref. 24]

types of modal logic have been developed, including *temporal logic* [Ref. 26] and *process logic* [Ref. 27].

The use of situational and modal calculi for the expression of reasoning introduces computational difficulties, however. In response, the STRIPS representation of actions was proposed [Ref. 28]. In STRIPS, a given state is represented by a conjunction of logical formulas. Events are represented by operators, each of which is composed of a *precondition*, and *add list*, and a *delete list*. This scheme for determining the ordering of successive states are embodied in a STRIPS rule.

Situational and modal calculi and STRIPS provided the foundation for what was to become the hierarchical approach to goal-based planning [Ref. 29]. Both are forms of logic, and both utilize rules to represent the inference process. The primary difference lies in the control of rule activation, also known as *chaining*. Chapter IV of this dissertation explains this concept further.

Real-time constraints presented a further challenge for these traditionally-structured systems. The deliberation required by the planners in these systems proved to be very time-intensive [Ref. 29]. In a hostile, dynamic world, replanning is frequently necessary and the welfare of the system often depends on the vehicle's readiness to act. Furthermore, such actions are often required in immediate response to the situation, leaving no time for deliberation. A number of systems were subsequently developed with these issues in mind [Ref. 30]. All were characterized by a departure from goal-based planning in favor of a high degree of reactivity. The extreme position was advocated by Brooks [Ref. 31], who proposed a bottom-up solution to the control problem involving a layering of task-achieving behaviors without regard to an internal *world model*. These control architectures stressed viability and robustness but at a cost of general problem-solving and reasoning [Ref. 29]. Demonstration of so-called "emergent" intelligence was provided by Brook's own robots [Ref. 32] and the Robart series of sentry vehicles [Ref. 33]. This approach formed the basis for the *subsumptionist*, or *behaviorist*, approach to autonomous vehicle control.

The value of reaction was not lost on the research community, however. Systems which had previously relied solely on hierarchical planning strategies began utilizing reactive procedures [Ref. 34]. This merging of hierarchical and behaviorist concepts has resulted in many instances of the class of *hybrid* control architectures. Each of these major approaches—hierarchical, behaviorist, and hybrid—are explored in more detail in this chapter. It is appropriate, however, to first discuss the capabilities of autonomous vehicles and the requirements expected of their associated control system.

C. AUTONOMOUS VEHICLE CONTROL REQUIREMENTS

Owing to the relative immaturity of the field and the complexity of the problem of control, research into mobile robots has had to address many issues in a multitude of areas. Robotics, computer vision, real-world modeling, sensor interpretation and integration, actuator and sensor control, path planning, navigation, plan execution, and system monitoring comprise an incomplete but representative list. The melding of this research to a physical vehicle has resulted in a remarkable test bed for new concepts, approaches, and technologies. The diverse disciplines involved have focused on the fundamental requirement of all autonomous vehicles: that they satisfactorily perform the mission given them while coping with a dynamically changing and unpredictable environment without human assistance. To meet this requirement, individual component technologies must exhibit levels of performance far exceeding the competence displayed by robotic systems operating in toy or highly specialized domains [Ref. 19].

Generally speaking, autonomous vehicles must acquire and maintain, through various sensors, a sufficiently detailed model of the operational environment to provide a basis for intelligent planning. Due to the intrinsic limitations and specialized characteristics of each sensor system, multiple sensor systems must be employed. This leads to a further requirement that the vehicle possess the capability of integrating these qualitatively diverse streams of data into a uniform and coherent form useful to the planning portion of the controller. Sensor error and noise, along with incomplete data due to insufficient sampling,

contribute to uncertainty. In addition, position errors are caused by vehicular motion and as a result of external forces. The deliberation process must be capable of accounting for these as well.

Obviously, the autonomous vehicle must have a means of accepting preplanned missions and precompiled maps as part of a pre-mission sequence. Relying only on information stored internally, the vehicle must be able to identify and classify objects observed by its sensors. An autonomous vehicle must also deal competently with problems of navigation and planning involving spatial reasoning, fault tolerance, obstacle avoidance, and replanning [Ref. 35]. Finally, the vehicle must provide for its own ultimate safety and, in circumstances where its warranted, the safety of humans and equipment relying on it.

Control software architects must wrestle with the efficient organization and integration of the systems designed to satisfy these many vital requirements. Initial designs approached this problem from one of two extremes: top-down with emphasis on deliberation, or bottom-up, where the emphasis was on sensing. Recently, researchers have seen the value of borrowing characteristics of both, resulting in improved performance. These approaches, along with a discussion of inherent strengths and weaknesses, are the subject of the next section.

D. SOFTWARE ARCHITECTURES FOR AUTONOMOUS VEHICLE CONTROL

Control software of autonomous vehicles is a field of study that is still in its infancy [Ref. 36]. As the underlying hardware and computational capabilities improve, and as cost and size constraints are lowered, the sophistication and complexity of the control systems for autonomous vehicles rises. Typically, the robotic vehicle and its associated control software have been developed in tandem. When a new autonomous vehicle is built, the control software system is likewise built from scratch. Design, implementation, and testing methodologies are, at this time, introduced in an *ad hoc* manner. In addition, the issue of mission reconfigurability is often totally ignored. These circumstances have resulted in a "low-lev-

el" view of software construction, accompanied by a lack of formal software engineering techniques. This situation has prompted research into better ways of organizing software in an attempt to address these difficulties. The result has been the emergence of the field of software architecture for the control of autonomous vehicles.

This research has resulted in a broad spectrum of software architectures. Some of these approaches are of a general purpose nature [Ref. 37][Ref. 38], while others are application-specific [Ref. 32][Ref. 39]. Only recently have the various approaches been categorized into four groups [Ref. 6], based primarily on the degree of deliberation utilized during system operation. These classes are hierarchical, behavioral, hybrid, and tool-based. Each has distinct strengths and weaknesses. Proponents of each approach point to specific instances in support of their chosen architecture. Ignoring the ill-defined tool-based category for the moment, all remaining approaches to control software for autonomous vehicles may be placed along the "architectural continuum", shown in Figure 1. At one end are the hierarchical control software architectures. At the other extreme lies the layered, or behaviorist, approach to control software. Hybrid architectures, which combine characteristics of the two "polar" camps, reside somewhere in between. This section describes these three main approaches in some detail and identifies current examples of each.

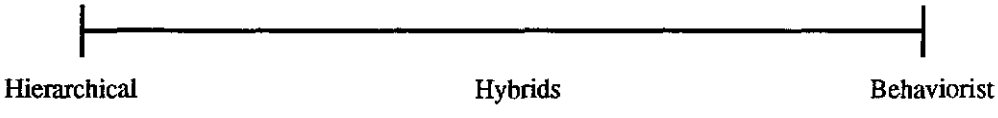


Figure 1. Architectures for Intelligent Control Software

1. Hierarchical Control Software Architecture

Systems of this type have a hierarchical structure [Ref. 40][Ref. 41][Ref. 42]. The problem of complexity is solved in a traditional manner whereby the task of autonomous

vehicle control is partitioned into successively less complex functional levels with high-level planning on top and low-level servo control at the bottom. Each level is subsequently implemented and then brought together to form the completed system. These systems are designed under the assumption that mission planning and mission execution algorithms should be abstracted to the top levels, leaving vehicle-specific functionality in the low levels. In support of sensory processing, a symbolic model of the autonomous vehicle's environment is maintained internally. This world model representation contains, at a minimum, the vehicle's current state as well as the current state of the environment. Some control systems, such as the NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM) [Ref. 43] and the Adaptive Suspension Vehicle (ASV) [Ref. 44], feature the use of hypotheses in task planning [Ref. 45]. A planning module can therefore simulate the outcome of some action before it has been executed. Thus, safety and optimization of planning is supported.

Each level in the hierarchy receives commands from the level directly above and sensory information from the level directly below it. This relationship can be referred to as a *command* hierarchy. Data elements at the lowest level are atomic. These atoms are "fused" with additional data into increasingly abstract data objects as they are passed to successively higher levels.

Another characteristic of this class of architectures is the increase in the update frequency as one moves from the top to the bottom of the hierarchy. At higher levels, this frequency is low, since problems are complex and the nature of data to be processed is typically symbolic. At lower levels, the problems to be solved are comparatively less complex and, while the amount of data available is significant, the formats used are conducive to rapid computation. At the lowest level, many of the primitive tasks can be partially or totally realized in hardware. Closely linked to this *temporal* hierarchy are the planning horizons of each level. Low update frequencies result in long planning horizons, and fast update frequencies require short planning horizons. The exact magnitude of these periods must often depend on experimentation, designer experience, and trial-and-error.

The control philosophy of the class of hierarchical architectures is naturally suited to an implementation of deductive reasoning called *backward chaining*. Backward chaining is a search strategy used by computers to solve problems of logic [Ref. 46]. In the context of autonomous vehicle control, the hierarchical control architecture executes its mission by determining if its ultimate goal has been satisfied. To do this, the goal must be divided into a sequence of simpler subgoals, each of which must be satisfied before the parent goal can succeed. Should a subgoal fail, an alternative reasoning path is tried. This process continues until either all the subgoals of the parent goal can be solved or all viable alternatives have been attempted without success. If these parent goals can be solved, then their parent goals can be solved in turn. This process continues until the initial (root) goal is satisfied, signifying mission completion. Automated reasoning as it relates to autonomous vehicle control software is discussed in detail in Chapter IV of this dissertation.

These systems suffer from a number of disadvantages. The rationale behind task decomposition is based on "best guess" rather than scientific formalisms. In practice, interactions between components of a control system are not identified until after prototyping and incremental development have occurred. In any case, a poor functional specification may not manifest itself until far into the development cycle. Change at this point may require extensive revisions and will certainly consume scarce resources, particularly if the logic of control is combined with other functional aspects of the software [Ref. 47]. Also, all modules within the hierarchy must be realized to some extent before the robot is capable of even the simplest of tasks. Furthermore, the existence of multiple hierarchies (command, temporal), viewed by different people at different stages of the project, may generate incompatibilities. To support the system requirement for robustness and fault tolerance, programmers must attempt to anticipate all possible scenarios in which the vehicle may find itself. These efforts often lack methodology and cannot be guaranteed to be complete. Finally, such systems are difficult to explain to the customers. Mission logic is often spread throughout the control hierarchy, making these systems difficult to reconfigure.

2. Behavior-based architectures

In contrast to the multilevel architectures, the task of autonomous vehicle control in behavior-based (or layered) architectures [Ref. 31] is viewed from a perspective of action rather than of deliberation. The overall behavior of the robot is developed incrementally. First, the desired task-achieving behaviors are identified. Next, they are grouped and ordered into levels of competence. Each level represents an informal specification of a class of behaviors exhibited by the robot regardless of the situation in which it finds itself. The classes of behavior at the lowest levels incorporate the simplest behaviors; successively higher levels of competence imply more complex specificity of behavior. Since it is important for the control system to be responsive to high priority goals while continuously servicing necessary low-level goals, each level of competence includes as a subset all underlying levels of competence.

With respect to the behavior-based control of an autonomous vehicle, levels of competence correspond to layers of the control system executing in parallel. New layers are added to an existing set to incrementally obtain a more competent robot. Each new layer is able to examine and, if required, alter data used by the next lower layer. The upper layer is said to *subsume* the lower layer, and together the layers achieve the level of competence associated with the top layer.[Ref. 31]

A distinguishing feature of the behavior-based architectures is the absence of a central intelligent source of control and an internal representation of the external world [Ref. 48]. Instead, data is distributed among all levels, and each level performs its own sensory processing. Additionally, commands and data are not passed from level to level as in the hierarchical architectures. By wiring together multiple layers of control, a robot has the potential to exhibit an intelligent global behavior. In effect, the intelligence *emerges* from the global interaction of multiple, unintelligent agents [Ref. 32].

Many of the functional requirements of autonomous vehicle control are satisfied by the behaviorist approach. Pursuit of multiple concurrent goals may be achieved with different layers working on different goals. The complexity and time requirements inherent

with data fusion are to a large extent avoided with each layer having access to pertinent sensors. The control system is robust in that, should the higher layers of control fail to produce results, lower layers continue to function in an expected manner. Finally, due to the incremental nature of the layering scheme, in theory at least, the subsumption architecture is readily extended by adding a new level of competence which provides the desired change in behavior.

Reasoning in behaviorist architectures, as it relates to mission control, is data driven; that is, computation starts with the existing facts and derives new facts (conclusions) from them. These systems chain forward from conditions known to be true towards states which those conditions imply. The process ends when either no additional facts are derivable from current facts or an accepting condition (goal state) is reached. This scheme suits the layered control entities which rely upon sensor-based data to determine their behavior. Data-driven reasoning and its forward chaining implementation is described in Chapter IV of this dissertation.

Unfortunately, the integration strategy upon which the subsumption architecture is founded is validated only by a process of trial and error. Also, it is impossible to verify the correctness and stability of the resulting control system. These drawbacks have serious implications for control of autonomous vehicles. From the user's perspective, an autonomous vehicle must behave predictably and reliably, especially given the sensitivity and danger of potential missions. With pure subsumption, these assurances cannot be given at this time.

Another issue not specifically addressed by Brooksian subsumption is the low-level control of the AV, namely the autopilot. With the laboratory robots Attila and Genghis [Ref. 49], both instantiations of pure subsumption control, the underlying stability of the system was assumed. That is, the subsuming behavior did not explicitly ensure that the robot would stay upright. If the robot flipped over trying to scale an obstacle, the experiment was terminated. Given the nature of the missions autonomous vehicles are called upon to complete, and the hostile environments they operate in, it is imperative that basic vehicle

stability is guaranteed. Therefore, control architectures for autonomous vehicles based on the behaviorist philosophy generally decouple low-level control from the layered behavioral control [Ref. 50]. The result is a two-level hierarchy, the lowest level of which is conventional feedback control.

A variation of behavior-based control has been advanced by Payton, et al., and is described in [Ref. 51][Ref. 52][Ref. 53]. This architecture, while still very much behaviorist, differs from the pure subsumption philosophy in the area of command conflict resolution. In subsumption, two commands from two related behaviors that are in direct conflict with one another are resolved by the highest priority behavior's commands completely overriding the other's. Compromise may be appropriate; unfortunately, subsumption does not allow compromise. By decomposing each behavior into small decision-making units, more flexibility is afforded the arbiter of conflicts. Additionally, by allowing behaviors to express preferences for a range of commands, allowance is made for the selection of commands that can simultaneously satisfy multiple goals (command fusion).

A further refinement of "pure" subsumption is presented by Mataric [Ref. 54]. This architecture incorporates a distributed world map representation into a homogenous reactive system. Although centralized world models are thought of as being incompatible with subsumption architectures [Ref. 48], any application requiring a solution superior to random walk must base its planning on a world model [Ref. 54]. The resulting control system remains fully reactive, however, due to the integration of representation with the layered behaviors. This is in contrast to hybrid systems which separate the control system into reactive and deliberative components.

3. Hybrid architectures

Because these two approaches are so fundamentally different, it initially appeared that little commonality could be abstracted in support of unifying the field. Instead, driven by the realities of implementation, projects which started out from purely hierarchical or behaviorist control perspectives have migrated away from their dogmatic roots. In their

place have emerged hybrid architectures borrowing features from both [Ref. 6]. Specifically, the hierarchical models failed to account for the long decision cycle times required by their planners. In many cases, this type of computation necessitated the use of platforms too large to be carried by the mobile robot. The behavior-based models, on the other hand, suffered from their lack of explicit intelligence. Reliance on emergent global behavior did not instill sufficient confidence to entrust expensive vehicles to their control [Ref. 55].

Hence, the field has experienced a move from the extremes to the center of the architectural continuum. Examples of behavior-based hybrids include State Configured Layered Control [Ref. 56], ISE Research's Layered Control [Ref. 50], and the Autonomous Land Vehicle Controller [Ref. 57]. Hierarchical-based hybrids include the Experimental Autonomous Vehicle (EAVE) controller [Ref. 39], the Remotely Operated Autonomous Robot Control System [Ref. 38], the Open Robot Controller [Ref. 58], and the Rational Behavior Model defined in this dissertation.

The goal of many hybrid models is to have task decomposition and explicit world representation at the higher levels and to employ behaviorist schemes at the lower levels. The middle level(s) are given the responsibility of providing the translation and coordination of commands and actions to and from the outlying levels. Hybrids have also striven to address the issues of mission reconfigurability [Ref. 55], implementation of automated reasoning [Ref. 59], real-time planning [Ref. 60], and module interfacing [Ref. 61].

The development of hybrid approaches, like those of their predecessors, have lacked a formal, mathematical basis upon which to build. This is a result of the complexity of the software and systems involved [Ref. 62]. While isolated algorithms can be analyzed using traditional methods [Ref. 63], and perhaps even individual levels can be described formally, complete systems simply do not lend themselves to theoretical examination. Instead, an iterative design and development approach is preferred, augmented by simulation studies, prior to full-scale experimentation [Ref. 64].

4. Tool-based architectures

This approach is favored by researchers that believe that no architectural principle can be stated at the present time. Experience alone is the driving force behind control software development in this approach. Unfortunately, in the field of autonomous mobile robots, there has not been sufficient time to construct a useful pool of experience. Vehicle technology, software complexity, and mission formulation are current research issues proceeding in parallel. Therefore, these architectures are much less applicable than those described above with respect to the current state of autonomous vehicle control software and, hence, will not be covered further.

5. Blackboard-based Control Architectures

Architectures employing blackboard data structures do not constitute a distinct group as defined by accepted taxonomies [Ref. 6]. However, many systems, both hierarchical and behavior-based, employ them as a means of inter-level or inter-process communications and data transfer [Ref. 17][Ref. 38][Ref. 43][Ref. 65]. Originally, blackboards were developed to provide communications between distinct rule bases of an expert system [Ref. 66]. Since then, however, the term blackboard has been applied to any globally-accessible data structure to which multiple processes may communicate by posting messages [Ref. 18].

As the size of the blackboard and the number of processes using it increases, however, so does the probability of undesired side effects caused by unexpected changes to variables. Concurrent access from independent, actively competing processes only exacerbates the problem. To avoid this possibility and therefore improve confidence in the behavior of the controlled vehicle, communications between the different levels of the software may be constrained in various ways. This usually results in an increase in lines of code and a more regimented programming style. However, this price is small compared to the resources needed to locate the subtle bugs which can result when unrestricted interaction between software modules is accomplished using global data structures.

E. TYPICAL AUTONOMOUS VEHICLES AND THEIR MISSIONS

A number of significant autonomous vehicle systems have been developed in recent years, primarily due to advances in processor capacity and miniaturization. Several of these systems, along with their philosophy of software control and system integration, are described next. This list is not meant to be all inclusive, and there are certainly additional programs which warrant discussion. Instead, the intent here is to highlight the current capabilities of these vehicles as a measure of "where the field stands" at this point in time.

1. DARPA/UUV

The joint DARPA/Navy Unmanned Undersea Vehicle (UUV) Program was initiated with the goal of demonstrating that UUVs could meet specific Navy mission requirements [Ref. 67]. To this end, DARPA developed a test bed UUV for new and existing technologies, including high energy density power sources, high data rate acoustic communications, and precision UUV navigation. The missions selected for these demonstrations were the Tactical Acoustic System, the Mine Search System, and the Remote Surveillance System.

Two UUVs were designed and built by the Charles Stark Draper Laboratory of Cambridge, Massachusetts. The vehicles are 36 feet in length with a 44 inch diameter hull and a weight, without payload, of 15,000 pounds. They are propelled by a 12 horsepower electric motor that can achieve a top speed of approximately 10 knots and a maximum operating depth of 1500 feet. A three-channel redundant computer system, running identical software, detects faults through a majority voting scheme. A full range of sensors supports guidance and navigation functions. Communication to the vehicle is possible over a RF link when surfaced and over acoustic telemetry when submerged.

Lockheed Missiles and Space Company has been awarded the contract to provide mission-level control software for the DARPA UUVs with a focus on fault-tolerant behavior. This architecture, called Autonomous Control Logic (ACL), is purely hierarchical and consists of three major components: the Data Manager, the ACL Controller, and the Model-

Based Reasoner (MBR). The Data Manager receives, processes, and analyzes sensor and status data for use by the MBR and ACL Controller. The ACL Controller communicates commands to an underlying vehicle control system under the direction of the MBR in support of plan execution, while providing for the safety and viability of vehicle and mission. Finally, the MBR comprises the "assessment" functions of evaluating the impact on the vehicle's capabilities due to unanticipated events and the generation of appropriate responses to those events. Servo and sensor control, guidance and navigation, and vehicle health and fault recovery are the responsibility of a related, but distinct, software system called the Vehicle Controller. ACL is being developed iteratively, and four prototype cycles are programmed. Formal acceptance of the system is to occur toward the end of 1994.[Ref. 68]

2. EAVE

The Marine Systems Engineering Lab at the University of Hew Hampshire has focused much of its research efforts on intelligent underwater systems. This effort is manifested in the Experimental Autonomous Vehicle System [Ref. 39], an AUV test bed. The current vehicle, EAVE III, consists of two buoyancy tubes, two battery tubes, and six DC thrusters capable of maneuvering the vehicle in four degrees of freedom. The EAVE III vehicle is 51 inches long, 41 inches wide, and 51 inches deep, and its overall weight is 1000 pounds. Its thrusters can provide a maximum speed of 1.5 knots and the vehicle has a depth rating of 500 feet.

A conceptual control software architecture has been proposed for EAVE and consists of four levels organized into in a hierarchical fashion. These levels are labeled, from highest to lowest forms of abstraction, *mission*, *environment*, *system*, and *real-time*, respectively. The abstraction mechanism used in this architecture is based upon process execution time. That is, determination of the placement of software functionality is determined by matching the time domain of the level which best corresponds to the execution time of the module implementing the function. Thus, the real-time level is responsible for sensory management and effector control; the sensor level deals with issues of vehicle integrity and

provides guidance functions; the environment level builds and maintains the world model and performs duties pertaining to vehicle navigation; and the top mission level handles mission-specific issues, replanning, and mission assessment. At this time, this architecture has yet to be fully implemented and tested either through simulation or on the EAVE vehicle.

3. ALV

The Autonomous Land Vehicle (ALV) was designed and developed by Martin Marietta Aerospace and is intended to be a test bed for research in autonomous mobility systems [Ref. 69]. Its dimensions are 2.7 meters wide, 4.2 meters long, and 3.1 meters high, providing the capacity to carry all power, sensors, and computer systems necessary to support autonomous operations. The ALV weighs approximately 16,000 pounds fully loaded yet is capable of traveling both on and off road. The vehicle has an eight-wheel drive, is diesel powered, and driven by hydrostatic transmission. A wide range of sensors is employed, including a video camera, a laser range finder, and wheel-mounted odometers.

A control software architecture was developed for the ALV by Hughes Artificial Intelligence Center [Ref. 57]. This hybrid architecture is organized into four levels, each containing planning and perception functions. At the highest level, the mission planner is used to define mission goals and constraints. These are passed to the next level, which maintains the world model and develops plans based on maps contained therein. The resulting route plan is then passed to the third level containing the local planner. The local planning module selects and monitors reflexive behaviors at the lowest level. It is at this level that reflexive behaviors are used as real-time operating primitives [Ref. 51]. Reflexive behaviors are independent of each other and execute concurrently; however, it is the responsibility of the local planner to partition the appropriate behaviors depending on the current environment. The ALV architecture has been field tested and was the first system to demonstrate obstacle avoidance in natural terrain [Ref. 70].

4. Ambler

The active exploration of other planets by mobile robots demands that they be fully autonomous. A manned mission, even to Mars, is highly unlikely in the foreseeable future. In addition, conventional teleoperation of robots is not practical due to the long time delays in signal transmission due to the extreme distances involved. An alternative solution would involve an autonomous mobile robot capable of safely navigating extremely rugged terrain while intelligently gathering materials and telemetry readings and returning them to earth for analysis.

Researchers at Carnegie Melon University have conducted a program to address the central problems of designing such a robot [Ref. 65]. The resulting prototype is called the Ambler, a vehicle consisting of a cylindrical body one meter in diameter and six legs mounted at different elevations above and around the body's central axis. Each leg is composed of shoulder and elbow joints and a vertical actuator to extend or retract the foot. The average overall height and width of the Ambler is 3.5 and 3.0 meters, respectively.

The proposed control architecture for the Ambler consists of a number of distributed modules with a centralized controller. Planning, sensing, and actuation are handled by the outlying modules. Identifying and prioritizing goals is centrally managed, however. Modules communicate with each other by posting messages to a message-routing table within the central control.

F. THE NAVAL POSTGRADUATE SCHOOL AUV

1. Capabilities and Characteristics

The Naval Postgraduate School Autonomous Underwater Vehicle is an unmanned, untethered, free-swimming robotic submarine. Its primary purpose is to support graduate student and faculty research in the areas of control technology, artificial intelligence, computer visualization, software architectures, and systems integration [Ref. 71]. The current iteration of the vehicle, shown in Figure 2, is 84 inches long and displaces 380 pounds. A maximum speed of two knots is attained by twin counter-rotating propellers

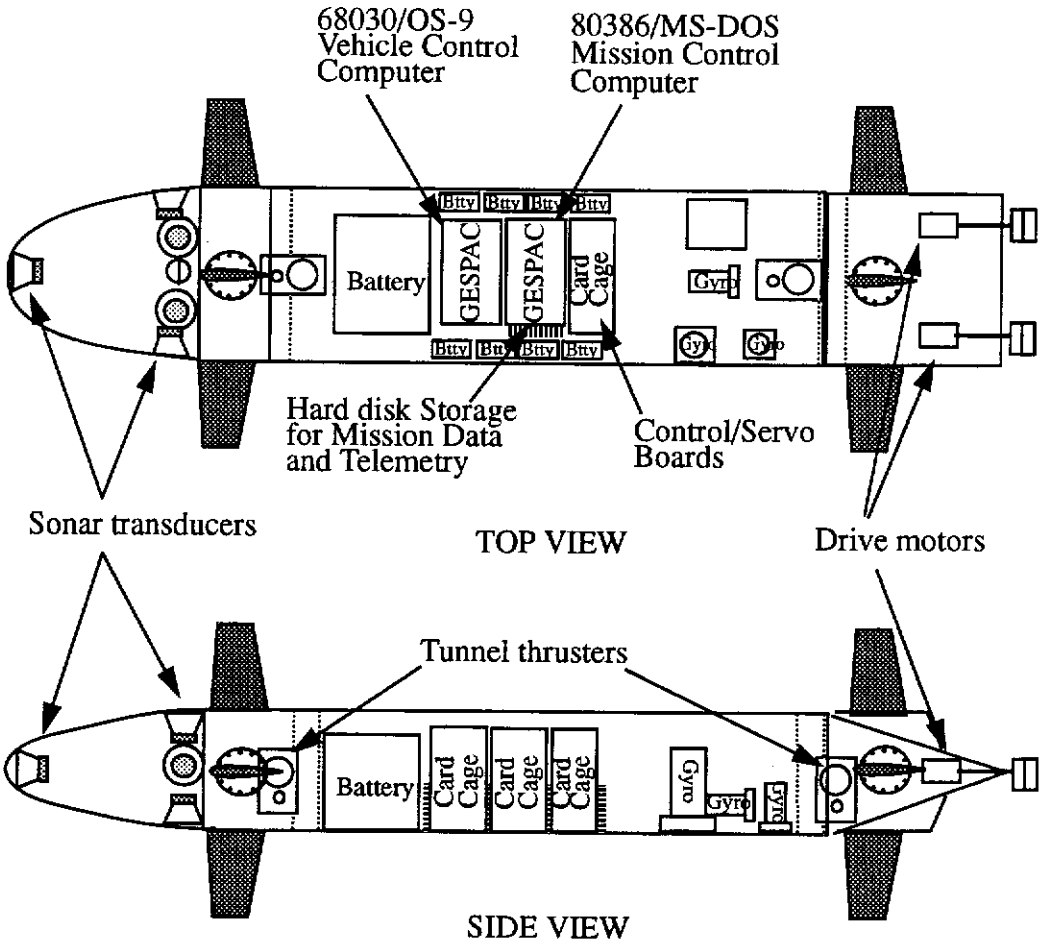


Figure 2. The Naval Postgraduate School AUV II

driven by DC electric motors. Eight control surfaces and four cross-body thrusters provide a high degree of maneuverability and motion control. The battery-based power system has an endurance of two to three hours. A dual-processor Gespac computational platform supports concurrent execution and separates high-level planning and navigation functions from low-level stability and actuator control functions. Miniaturized, low-power ultrasonic sonar, inertial navigation, and global positioning systems are also available. The small size and low cost of the NPS AUV are ideal for the support of research on a wide range of shallow water missions, including search, mapping, surveillance, and intervention activities [Ref. 72].

Software control of the NPS AUV is provided by an instantiation of the Rational Behavior Model (RBM). RBM is defined in Chapter V of this dissertation; the NPS AUV instantiation and results from laboratory simulation studies are presented in Chapter VI.

2. Simulation Facilities

Due to the high expenditure of resources involved in preparing the NPS AUV for in-water experiments, and as a means of verifying and validating software prior to integration with the vehicle, extensive simulation testing is done. This process allows the system developers to observe the impact of a new or modified module on the total system. To this end, a detailed graphics simulation of the vehicle and its swimming pool environment has been developed. The simulation, besides rendering a visually accurate reproduction of the physical vehicle, includes hydrodynamic coefficients, vehicle mass characteristics, and variables for vehicle dynamics in depicting the AUV's motion. By measuring the elapsed time between frame updates, the calculations simulate real-time motion.

The graphical simulator acts as both the physical vehicle and its servo level control system. Higher level control systems reside on separate computers and communicate heading, speed, depth, and mode commands to the simulation over an ethernet connection. For the purpose of this dissertation, the higher levels of control were hosted on Sun SPARCstations.

An extension of the simulation facilities for the NPS AUV is currently under development. Specifically, the integrated simulator effectively networks a three-dimensional graphical simulator with an exact replica of the NPS AUV computer, software development workstations, and assorted support resources [Ref. 73]. Once completed, this system will supercede the current simulator, since the lowest level control software will be that which actually controls the vehicle actuators and sensors. This allows for the testing of hardware as well as software components. The graphics workstation provides accurate representation of vehicle dynamics while permitting experimental evaluation of developmental software and post-mission reenactment from recorded telemetry.

G. SUMMARY

This chapter has provided an overview of the field of autonomous vehicles and their control. From rather humble beginnings, autonomous vehicle technology has advanced to the point where increasingly difficult and complex applications are deemed achievable. To attain the level of expectation now associated with autonomous vehicles, the requirements placed on the design of such a system are so numerous as to be nearly overwhelming. From a computer science perspective, what is most urgently needed is a comprehensive software architecture that provides the necessary organization and effectively manages the complexity of these various systems. Initially, two general approaches were taken toward the solution of this problem: the deliberation-based, hierarchically-structured architecture and the reflexive-based, behaviorally-layered approach. Experience has shown that neither provides the proper combination of intelligence, response time, and robustness required for the successful operation of mobile robots in uncertain, hostile environments. This has resulted in the emergence of a hybrid class of architectures characterized by higher-level planners utilizing internal world models interfacing with lower-level, reactive control systems.

Most, if not all, software architectures for the control of autonomous vehicles reported in the literature remain in a conceptual state or only partially realized. Many admit to the uncertainty of how to proceed in the development of the "intelligent" portion of their re-

spective architectures [Ref. 39][Ref. 56]. This is partially due to the emphasis of research placed on these projects and to the technical backgrounds of the personnel involved. Instantiation of software systems involves careful consideration of the computational domains involved. This directly influences the choice of the programming paradigms and languages used in the development of the system. The next chapter of this dissertation provides the reader with a review of the alternative paradigms, the strengths and weaknesses of each, and the applicability of each with respect to software architectures for the control of autonomous vehicles.

III. PROGRAMMING PARADIGMS AND LANGUAGES

A. INTRODUCTION

Once the desired capabilities of an autonomous vehicle have been identified, efforts must be turned to the design and, ultimately, the implementation of the system. With respect to software, these phases in the development cycle can proceed smoothly for systems of small complexity and size. Generally, applications consisting solely or primarily of numerical computation are ideally suited for an imperative programming approach using a commonly available imperative language. Engineers, mathematicians, and scientists deal with problems of this type and are, therefore, most comfortable with this paradigm. However, intelligent control of autonomous vehicles introduces a wider range of problems to be solved. Some, like navigation and servo control, lend themselves well to imperative solutions. Automatic reasoning, on the other hand, involves the encoding of human knowledge. This information is decidedly non-numeric; therefore, a different programming paradigm is called for, one better suited to the representation and manipulation of symbolic structures.

Software organization and data management are also considerations that must be part of the overall design process. The method by which modules communicate, how data is passed and stored, constraints on time and space, concurrency, and distributed processing are all issues which should influence the choice of programming language(s) used in an implementation. Additionally, practical issues, such as the availability of resources and the experience of personnel, will certainly come into play.

Theoretically, the expressive power of most computer languages are the same. Any problem that can be expressed in one can be expressed in another [Ref. 74]. From a practical standpoint, however, languages differ dramatically with respect to the class of problems for which they were designed. Each has a domain of problems in which it excels while, in

other contexts, the same language may prove to be unacceptably inefficient or unwieldy. For this reason, there is probably no "best" single language for applications involving a mix of problem domains. The following discussion is meant to aid the designer of autonomous vehicle systems in the informed selection from the wide range of candidate languages.

B. IMPERATIVE PROGRAMMING

The class of imperative languages have their origins in the earliest computing machines. These computers had little storage and were very slow. Programmers were required to focus much of their energies on code optimization. Since memory typically was provided in the form of a rotating drum, this process entailed the calculation of instruction location based on the distance traveled by the drum during the execution of the previous instruction. Also, there existed no facilities for indexing or address modification, important features of the *von Neumann* architecture upon which most modern computers are built [Ref. 75].

The complexity of programming these machines led to the development of *flow diagrams* and later to the now well-established *flow chart*. These design tools assisted the programmer by describing explicitly the flow of control through the program; that is, the points where branching, looping, and sequencing occurred were represented by appropriate symbols. Program development was made even simpler as a result of the introduction of low-level language *interpreters*. By writing programs in a form other than machine code, the programmer could concentrate on the application at hand and leave the more tedious and error-prone activities to the interpreter. [Ref. 74]

A drawback of the new interpreters was that they ran slowly. Each instruction was translated into an intermediate form which could then be executed by the computer. However, decoding each instruction added a great deal of overhead to program execution. With the introduction of floating-point hardware by IBM in 1953, the overhead required by the interpreter began to dominate the total execution time of the program. For this reason, interpreters fell from favor and were steadily replaced by *compilers*. The process of compilation involves selecting pre-written and tested subroutines from libraries and assembling

them under the direction of the user's code into an executable program. Translation and decoding was performed only once, at compilation time, resulting in programs that executed much more quickly than interpreted programs.

At this time, John Backus of IBM recognized that it was becoming more expensive to design and debug programs than it was to run them. He concluded that programming costs could be decreased only through the use of a system that could recognize conventional mathematical notation and generate code equivalent in efficiency to that produced by a good programmer. This effort led to the development of the Formula Translating System, FORTRAN [Ref. 76].

The original FORTRAN, like its various dialects developed through the years, was characterized by the distinction made within its programs of a declarative part, describing data areas and their initial values, and an imperative part, containing the commands to be executed during the running of the program. *Declaratives* state facts about the program which are used at compile-time. *Imperatives*, on the other hand, describe actions which the program obeys at run-time, such as algebraic computation, assignment, comparison, and branching. This distinction is characteristic of the class of *imperative programming languages*, a class that includes, besides FORTRAN, PL/I, BASIC, Pascal, C, and Ada.

Development of these other languages was driven by three primary factors. First, FORTRAN's lack of character manipulation facilities limited its effectiveness in applications that were not strictly numeric. Second, FORTRAN was, for the most part, based on the hardware architecture of a particular family of machines and, therefore, lacked syntactic regularity and consistency. Third, FORTRAN programs relied heavily on the GOTO statement as a control-flow mechanism. As FORTRAN programs increased in size and complexity, they often suffered from "spaghetti-like" structuring that was difficult to understand, debug, and maintain. This phenomenon caused a reaction within the field of computer science that resulted in the introduction of *structured programming*, a body of programming methods intended to foster easier and more reliable programming [Ref. 77].

C. FUNCTIONAL PROGRAMMING

Imperative programming depends heavily on the assignment statement and a changeable memory for program accomplishment. Most imperative languages can be viewed as consisting of a variety of mechanisms for routing control from one assignment to another. In *applicative* programming languages, however, it is the applying of functions that is central. Hence, these languages are also referred to as *functional* languages. The functional approach is characterized by its lack of a destructive assignment. In other words, a pure function does not produce side effects; rather, it returns a value based solely on the values of its input parameters. Repeated calls to a pure function with the same arguments will always produce the same results. This phenomenon is called *referential transparency* [Ref. 78].

Functional programming is also characterized by its lack of control structures. By the judicious construction and application of functions, a GOTO statement is not required. This is made possible because arguments to functions may themselves be functions. For example, the IF-THEN-ELSE conditional control structure available in conventional languages is written functionally as the application of a "condition" function (*cond* in LISP) to a number of parameters representing the various alternative branches. Even function definition is accomplished by invoking a function (*defun* in LISP) with arguments representing the name of the function, a formal argument list, and the function's body[1]. One advantage of programming this way is its simplicity in that all components of the program are represented and evaluated in the same manner [Ref. 79].

LISP grew out of the need within the field of artificial intelligence to represent complex interrelationships among primarily symbolic data [Ref. 80]. This computation is accomplished by allowing the manipulation of lists in ways comparable to the ways imperative languages manipulate numbers. Lists can be compared, passed as parameters, constructed, and taken apart. In fact, LISP data, functions, and, by extension, its programs are all written as lists (known as *symbolic expressions* or *S-expressions*).

---

1. To be precise, this involves binding the function name to a value; hence, *defun* is not a pure function.

Viewed as an applicative language, there are only two control structures in LISP: the aforementioned conditional expression and recursion. The conditional is used to define logical connectives such as AND and OR. As a result, the operands of a logical relation are evaluated sequentially as they are encountered, from left to right. As soon as one of its arguments evaluates to FALSE, the *and* function immediately returns FALSE. Similarly, the *or* function sequentially evaluates its arguments until a TRUE is returned. Remaining arguments are ignored. This interpretation of the logical connectives is known as *lazy* evaluation and is opposed to the strict evaluation of Pascal which first evaluates all logical sub-expressions before evaluating the full expression. Strict evaluation can lead to errors if one of the sub-expressions is undefined [Ref. 81].

The other control structure provided by purely functional LISP is recursion. All forms of iteration including the imperative WHILE, REPEAT-UNTIL, and FOR loop control mechanisms are performed using recursion. This is possible because recursion relies on reducing the unsolved cases of a problem to a form for which an answer exists. Usually this involves a function calling itself with successively shorter lists as parameters. Recursion can also be used to "map" or apply a function to every element of a list and to filter, or extract, certain elements from a list. This is accomplished without the need for indices, control variables, or explicit bound declaration. So as to not violate the functional tenet of side effect avoidance, these functions manipulate copies of their inputs rather than the inputs themselves.

There are other practical benefits of functional programming. Data structures, in the form of lists, are treated as units, thus isolating the programmer from the mundane and error-prone details of explicitly manipulating these structures and managing their storage. Instead, these responsibilities are left to the interpreter and host machine. LISP thus supports programming at a higher level of abstraction than does the imperative approach [Ref. 74]. Another advantage of LISP is that it can be interpreted. Interactive programming at a terminal with rapid response is invaluable in the step-wise development of software. Debugging and program modification are not only possible "on the fly" but are greatly simplified

compared to the effort and resources typically required by compiled languages. Since LISP programs are themselves lists, it is convenient to manipulate LISP programs using other LISP programs. As a result, a broad spectrum of software tools have been written in LISP, including program translators, compilers, interpreters, and text editors.

Although LISP is the closest thing to an applicative language available for general use, it is not purely functional because it allows for the definition and manipulation of data. Functions that alter the state of the computer while computing a result are called pseudo-functions or procedures. Many of these "destructive" functions evolved from attempts to increase the performance of LISP. Although these functions remain part of the language definition, the current state of computer performance has made performance considerations moot. Therefore, the use of destructive functions is discouraged so as to avoid undesirable side effects that may result.

In addition to providing structured programming control forms, LISP has adopted the features of object-oriented programming as a result of the ANSI standardization of Common Lisp. The Common Lisp Object System (CLOS) provides to the software system designer the benefits of objects and classes (which are described in detail in section E of this chapter) while retaining the procedural abstraction and encapsulation of "traditional" LISP [Ref. 82].

D. LOGIC PROGRAMMING

One advantage of LISP is that it hides much of the lower-level manipulation and management of data from the user. This is important because the less detail the programmer is faced with the less chance there is for error. Another way of saying this is that a language that handles concepts at a higher level than another is also "less procedural". The user of a high level language can devote more effort to what is to be done and less on how to accomplish it. In the extreme, a non-procedural language would represent only the desired goal of the computation; the computer would then be left to determine how the goal was to be achieved [Ref. 74].

In many respects, this approach to problem solving is similar to automated problem solving in the field of artificial intelligence. Theorems involve the statement of a hypothesis followed by a proof linking what is known to be true to what is to be proved. The proof relies on the axioms and tenets of formal logic and the formulation of exact deductive reasoning about them [Ref. 83]. Logic programming makes use of the observation that applying standard deduction methods often has the same effects as executing a program [Ref. 84]. Thus, programs express propositions that assert the existence of the desired conclusion. It is the job of the theorem prover to construct from the set of premises this result and, hence, prove its existence.

The programming language Prolog (Programming in Logic) attempts to automate this process. Prolog programs consist of clauses, of which there are three types: facts, rules, and queries. Rules define the problem domain and facts make assertions about the domain that are known, or assumed, to be true. The rules and facts comprise a Prolog database. A query is a statement used to extract information from the program. Taken together, the three parts of a logic program resemble the statement of a mathematical theorem.

The general form of a clause is <head> :- <body>. If the head is omitted, the clause is a query; if the body is omitted, the clause is a fact. A clause with both head and body is a rule. The head and body are composed of relationships, each of which is either an application of a predicate to one or more terms, or an atom. Prolog allows at most one relationship in the head, a form referred to as a *Horn* clause. An explanation of why this is so is deferred to Chapter IV of this dissertation.

In pure logic programming, because control and logic are separated, the ordering of the clauses is irrelevant to the execution and results of the inference process. The program's logic is based solely on logical relationships between clauses instead of their physical relationship. Prolog, in the interest of efficiency and determinism, includes control mechanisms that greatly influence how programs are written and organized. First, Prolog uses backtracking as a way to explore alternative paths to a goal. If a dead end is encountered, the search simply continues from the last decision point. Second, by its definition Prolog

uses a depth-first strategy when searching through a database. It becomes the responsibility of the programmer to arrange the clauses and subgoals of the program so as to avoid the pitfalls of depth-first search. Third, as a consequence of these two characteristics, Prolog includes the cut (!) mechanism designed to preempt backtracking and halt the search for additional solutions.

Determining if the query is a logical consequence of the program (that is, it can be "proved" by the rules and facts available) is the responsibility of the Prolog inference engine. This is itself a program, separate from the user's program, that performs the task of deriving inferences from the given set of clauses. If the response to a query is yes (or TRUE), the goal is provable from the available assertions. If, however, the system returns no (or FALSE), the query has not been proven false; rather, it could not be deduced from the existing premises. This inference based on lack of knowledge is otherwise known as the *closed-world assumption.* It is possible to ask existential queries in an attempt to find specific solutions. These queries involve one or more variables which are bound, through a process called unification, to appropriate values. The system will continue to search the set of clauses, if commanded, until all possible solutions are generated.

Prolog has very few built-in data types and essentially no data structure constructors. Instead, data structures are defined implicitly by the clauses in the program and manipulated by matching the data against existing expressions. As in functional programming, recursion is used in place of the procedural control structures of imperative languages. Unfortunately, attempting to apply Prolog in a purely logical way, say for integer arithmetic, is intolerably inefficient, particularly since the computer can implement this directly [Ref. 74]. By including "procedural" predicates and functions to improve performance, however, logical properties are compromised [Ref. 85].

E. OBJECT-ORIENTED PROGRAMMING

Partly as a reaction to the "software crisis" of the 1970s, when the costs of producing software were increasing apparently without bound, languages characterized by their em-

phasis on data abstraction were developed[2]. Specifically, these languages, of which Ada is a significant example[3], provided an encapsulation facility supporting the separation of specification and implementation, information hiding, and strictly controlled access to function and data declarations. In addition, the concept of the *class* (generic type in Ada) was included in these languages, from which multiple instances of a data structure could be created. The purpose of this feature was to support software reuse, which in turn can be expected to enhance program efficiency, readability, and maintainability while reducing associated cost.

These same motivations lie behind the development of object-oriented programming languages. These languages allow the definition of classes from which abstract data structures called objects may be instantiated. Objects typically contain a set of variables that may be manipulated only by specified operators defined for that purpose. These operators are called *methods* and are invoked upon receipt of a message. By restricting access to the methods and variables of an object, the object's internal structure is said to be *encapsulated*, and any modifications made to this structure will not affect programs using that object [Ref. 86].

One of the features that distinguishes object-oriented programming languages is the code-sharing mechanism called *inheritance* [Ref. 86]. With inheritance, a new class can be designed from an existing one by specifying only those variables and methods that differentiate the two. The newer subclass is said to inherit all the features of the parent superclass. Of course, this includes those items the parent class may have inherited from its superclass. The subclass relationship between classes defines a *class hierarchy* with inheritance operating over "a-kind-of" or "is-a" links joining subclasses and superclasses.

---

2. Data abstraction manifests itself in the Abstract Data Type (ADT), a set of data values and operations on those values. The representation of the data is hidden within the module. Access and manipulation of the data is accomplished only through the procedures provided by the module to the user.

3. As will be emphasized shortly, Ada is object-based rather than object-oriented due to its lack of provision for class inheritance. However, Ada's support of ADTs and encapsulation is still pertinent to this discussion.

Objects may contain variables that represent other objects. In this case, the containing object is a composite object. The variable object may be implemented in one of two ways: as a dependent object or as a subobject. A dependent object cannot be created unless the composite object is first created. Similarly, the dependent objects are destroyed when the composite object is destroyed. These objects share a "part-of" or "has-a" relationship. Subobjects, on the other hand, are created independently of the composite and are subsequently linked through a pointer stored in the corresponding variable. Subobjects are not automatically destroyed upon termination of its composite object. In addition, subobjects may be shared between any number of composite objects [Ref. 87]. Such objects exhibit a *uses* relationship [Ref. 88]. The relationship between composite, dependent, and subobjects forms the basis for object (or composition) hierarchies.

Based on the definition for object-oriented programming proposed by Wegner, a language is object-oriented if it supports objects, classes, and inheritance [Ref. 89]. Languages which feature the first two characteristics but lack inheritance are called object-based. Ada, because it has no provision for inheritance, falls into this category.

Object-oriented features, when added to an existing language, results in a *bolted-on* language; languages designed with object-oriented principles in mind are called *built-in* [Ref. 87]. Examples of bolted-on languages are C++ [Ref. 90], CLOS [Ref. 91], and Classic-Ada [Ref. 92]. Built-in languages include Smalltalk-80 [Ref. 93] and EIFFEL [Ref. 94].

Part of the confusion associated with the object-oriented programming paradigm results from a lack of generally accepted terminology [Ref. 95]. The most significant differences lie in how certain mechanisms are implemented within the language, such as the assignment and persistence of class variables and the realization of multiple inheritance. As the field matures, a more precisely defined object-oriented paradigm should evolve, reconciling current language differences to a greater degree than at present.

F. CONCURRENCY AND REAL-TIME ISSUES

For the most part, the problem domains for which the above classes of programming languages were developed excluded applications where multiple processors were available or where execution times were constrained by fixed deadlines. With the advent of fourth generation programming languages[4], language features taking advantage of potential parallelism have been included. The ability to specify concurrent tasks alone is not sufficient, however, to meet the requirements of real-time systems. Additional considerations must be accounted for by those who design and implement such systems. The purpose of this section is to delineate these issues and discuss various approaches to the solution of each.

1. Concurrent Programming

Although research into concurrency is ongoing for all classes of languages, those discussed thus far are usually executed on a single processor. That is, programs written in these languages typically have a single active thread of control, in which one program segment is executed at a time. In many cases, the performance of these programs could be improved if independent computations, also known as processes, were run in parallel. The identification of notations and techniques for expressing potential parallelism and for the solving of the resulting synchronization and communication problems is called *concurrent programming* [Ref. 96].

In general, concurrency within a program can be achieved either by specifying a standard interface to a multitasking operating system or by expressing it in the language itself. Operating systems like UNIX support many executing processes in concurrent fashion by allocating to each use of the processor in rapid succession. The effect of this "time slicing" is to emulate true parallelism.

The focus of this discussion is on the second implementation. Whereas earlier generation programming languages like FORTRAN and C are purely sequential, and appli-

4. The reference here is made to fourth-generation programming languages as defined by MacLennan in [Ref. 74]. This category includes those languages which provide the data abstraction facility of encapsulation and control constructs supporting concurrency.

cative languages like LISP contain no provision for explicitly directed concurrency, the so-called fourth generation programming languages, including Ada, include distinctive control structures for the expression of parallelism within their definition [Ref. 74]. Ada, for example, identifies separate threads of control as tasks within the main program. Significantly, concurrent programming languages provide abstraction mechanisms to isolate the programmer from the details of how the parallelism is actually implemented. Thus, an Ada program requires no modification should the underlying computer change from a single to a multiple processor architecture [Ref. 96]. If a single processor is available, task execution will be interleaved. If, however, a processor is available for each task, they will execute in parallel; that is, the concurrent tasks will be simultaneously active.

Concurrent tasks must communicate with each other in order to synchronize their activities or exchange data. One way to accomplish this is to employ a shared (global) area of memory accessible by the relevant tasks. In a distributed system, however, a centrally accessible memory may not exist. In this case, processes must communicate through message passing. Using Ada again as an example, a task wishing to communicate with another invokes an entry call recognized by the target task. Synchronization is accomplished via the Ada *accept* statement. Once the called task acknowledges receipt of the message, the two tasks proceed to execute concurrently. Note that this is unlike a procedure call, where the calling program is suspended until the called program completes its processing, returns the results, and is deactivated. Should a task reach an accept statement before the corresponding message is sent, the task suspends itself until the message arrives. Likewise, if a task attempts to transmit a message before the receiver is ready, it will be suspended. In Ada, this coordination mechanism is called a *rendezvous*. Ada also provides for the conditional acceptance of alternative rendezvous through the use of the *select* statement. Other concurrent languages, such as OCCAM, Modula, and Mesa, provide similar functionality but employ widely varying mechanisms for its attainment [Ref. 97].

The problems associated with the sharing of data between concurrent objects is of significant interest within the object-oriented programming community. Class variables are

typically shared between all instances of the class, and few object-oriented languages provide for the mutual exclusion of concurrent objects vying for access to these variables. At the individual object level, instance variables may also experience mutual exclusion problems if concurrently executing methods accessing the same variables are present. Additionally, the consistency of class variables may be violated if a class definition is replicated on multiple nodes of a distributed system. Modification of one such variable at one site requires that every copy be similarly modified.[Ref. 98]

When writing a concurrent program using a concurrent programming language, it is the responsibility of the programmer to identify the activities which may done in parallel. This is difficult for several reasons. Most "tried and true" design and program methods available were created for sequential programs. Parallel systems must be approached from a totally different perspective. Partitioning problems into independent threads of control is a skill developed through experience. Insuring the correctness of concurrent programs is also more difficult than for sequential programs. A mature body of proof theory has been built around traditional algorithms [Ref. 63], and the field of software testing has developed a wide assortment of methodologies whose purpose is to instill confidence in systems implementing these algorithms [Ref. 99]. For the most part, however, these techniques do not apply to concurrent programs because simultaneously executing processes can interfere with each other in subtle and apparently random ways. These problems occur when resources are shared and fall into one of two general areas: mutual exclusion and deadlock [Ref. 100]. Although the issues of program correctness for both sequential and concurrent programs are important and relate to the work presented in this dissertation, they are beyond its scope. Additional information may be found in [Ref. 101] and [Ref. 102].

2. Real-Time Support

Improved performance is the prime motivation for concurrent programming [Ref. 103]. Certain systems, particularly those providing critical monitoring and control functions for a larger system, require the level of performance only true parallelism can yield.

In addition to managing concurrent threads of control, some systems must insure that each task meets specified deadlines [Ref. 104]. These systems are known as *embedded* or *real-time* systems, because they must respond to real-world requests in a timely fashion. Real-time, as implied in this context, simply refers to the timing constraints imposed on the system by the application.

Real-time systems are categorized as either *hard* or *soft*. In a hard real-time system, the output of the system must not only be computationally correct but temporally current [Ref. 105]. Systems which operate under time-based constraints but for which a late result is acceptable are called soft real-time systems. Given that their safety depends on timely responses to dynamically changing circumstances, the lower, vehicle-control level of the software architecture of autonomous vehicles typically falls into the hard real-time category.

A common trait of real-time control is the substantial amount of data that must be sampled and processed for use by the embedded system. As a result, the language chosen to implement real-time control must have the ability to manipulate floating point numbers quickly and with a high degree of precision [Ref. 106]. System response time is dictated by the timing constraints placed on the controller as well as the overall efficiency required of the system. Three approaches are taken to meet the required level of performance: increasing processor speed and memory, optimizing code, and concurrent computation using multiprocessors. In the future, gains of the magnitude required by ever more complex systems will be realized only through concurrent processing [Ref. 107].

Because of their numeric orientation and the availability of optimizing compilers, general purpose imperative programming languages are often used to in real-time applications. This is something of a contradiction, as these languages do not permit explicit expression of timing requirements. Nevertheless, performance specifications can often be met through a combination of code "tuning", load balancing, memory management, and scheduling policy [Ref. 108]. The major disadvantage to this approach is that it results in a rigid and delicate system not amenable to modification.

Due to the sheer complexity of real-time systems and the consequences of system failure, this process must be replaced with design methods encompassing system flexibility, robustness, and verification. Reliance on the operating system to generate schedules that guarantee performance is one potential solution [Ref. 109]. From the programming perspective, several specialized languages have been developed that provide constructs defining temporal concepts such as absolute time stamping, during, period, and priority of processes. Examples include Flex [Ref. 110], ESTEREL [Ref. 111], and Ada 9X [Ref. 112]. Object-oriented languages and operating systems designed specifically for use in real-time applications have recently appeared; an overview of these systems is found in [Ref. 113]. Another promising approach under investigation is the use of computer-aided software tools for the design and development of real-time systems [Ref. 114]. Interested readers are directed to [Ref. 105] and [Ref. 109] for excellent overviews of the issues involved.

G. SUMMARY

The system designer of a complex software architecture is faced with many programming paradigms and languages from which to choose. An implementation should, however, be driven primarily by the nature of the problem to be solved. If a varied spectrum of problems exists, it may be appropriate to select a mix of languages, each suited to a particular application. Autonomous vehicle control serves as a prime example for this approach.

One problem to be faced in such a project is the representation of knowledge within the intelligent controller. LISP was mentioned as a good language for symbolic manipulation. Prolog is based on predicate calculus and deduction. Although neither is ideal for the construction of automatic reasoners [Ref. 115], both offer viable approaches. In addition, another group of languages has been developed: rule-based. These topics will be covered extensively in the next chapter.

IV. LOGIC AND REASONING

Autonomous vehicles, by definition, are designed to operate without human intervention. This is in contrast to Remotely Operated Vehicles which contain a "man in the loop". Since a human is not available to make decisions on the spot, the autonomous vehicle must have the capacity to reason. Much research done in the field of Artificial Intelligence has striven to attribute this ability to machines [Ref. 116]. To impart knowledge to an autonomous vehicle, facts and rules describing the problem domain in which the vehicle will operate must be placed into a form easily manipulated by a computer. This manipulation is accomplished in an orderly fashion by a reasoning mechanism using a particular control structure. The mechanism is an *inference engine*, and the control structure is the type of *chaining* employed by the inference engine. This chapter begins with a review of computational logic, upon which all forms of automated reasoning are built. Next, the concepts of *goal-directed* and *data-driven reasoning* are discussed, as well as the implementation strategies of backward and forward chaining. Chaining is used by an automated reasoner called a *rule-based system*, and the next section describes them. Lastly, representations of the search space associated with rule-based systems, namely AND/OR goal trees and State-Transition Diagrams, are discussed.

A. INTRODUCTION

Intelligent software systems are also known as *knowledge-based systems* [Ref. 46] and are made up of two parts: (1) a knowledge base consisting of facts, rules, concepts, and theories, and (2) an inference mechanism, which examines (searches) the knowledge base in an orderly manner and answers questions, reasons, and draws conclusions. Automated reasoning describes the behavior of such an intelligent system. If the knowledge encoded within the knowledge-based system represents the expertise of a human expert in a particular domain, the system is termed an *expert system* [Ref. 115]. The knowledge representation

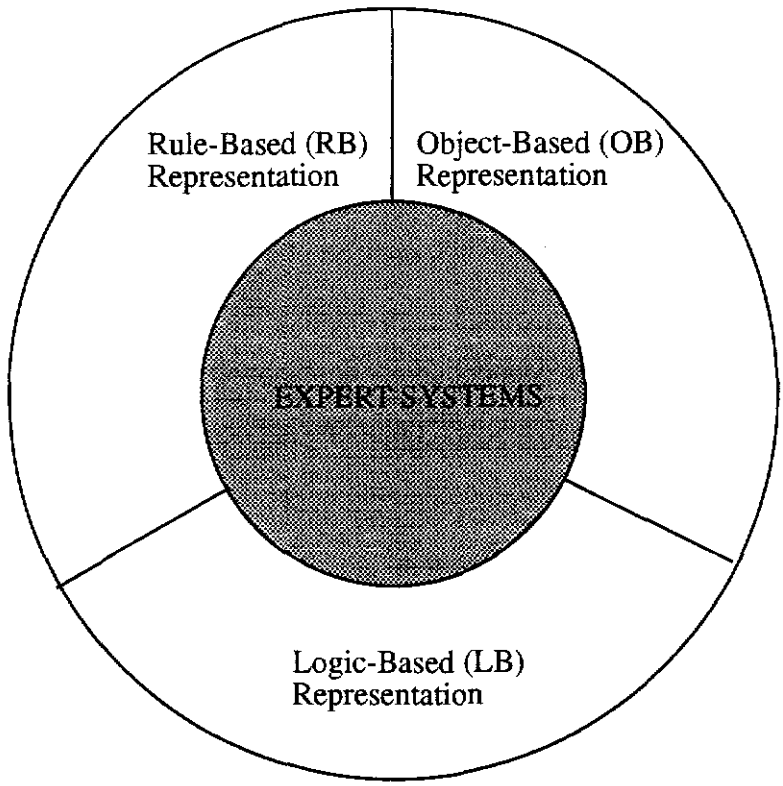
used by all these systems is generally expressed in symbolic, as opposed to numeric, form. Various knowledge representation formalisms exist, although practical realizations typically rely on some combination of predicate logic [Ref. 117], production rules [Ref. 118], and structured objects [Ref. 119]. Figure 3 depicts the three most common knowledge representation schemes, the inference mechanism employed by each, and their relationship to expert systems. As can be seen in the figure, expert systems typically display characteristics of more than one approach.

Rule-based expert systems that are capable of solving mission-specific problems are used in the construction of the Rational Behavior Model; thus, this chapter will focus on them. However, rule-based systems are founded upon the precepts of mathematical logic, and it is therefore necessary to introduce this topic first.

B. COMPUTATIONAL LOGIC

A discussion of the topic of automated reasoning must begin with an overview of the basics of formal *logic*, the science of correct thinking. What is presented here is intended to be an overview only, with a purpose to provide the context for what follows. For more thorough coverage, readers are directed to [Ref. 120].

Formal logic is a language consisting of expressions and a set of grammatical rules which, when applied, can determine the truth or falsity of expressions. Logic is formal because the meaning of an expression is defined strictly by its form. In addition, logic is precise about the conditions under which expressions evaluate to true or false. The body of study surrounding logic is often called a *calculus* because all results drawn from the application of a rule depend solely on the expressions themselves and not upon extraneous ideas or intuition. Of interest here is the simplest logic formalism, *propositional calculus*, and the more general logic system called first order *predicate calculus*.



Knowledge representation:

RB - production rules
LB - predicate calculus
OB - structured objects

Inference mechanisms:

RB - Chaining
LB - Resolution
OB - Inheritance and procedures

Figure 3. Knowledge Representation and Expert Systems

1. Propositional Calculus

The propositional calculus, also known as Boolean algebra [Ref. 121], is a language consisting of a symbolic notation and rules for the manipulation of these symbols. Taken together, this language can perform sequences of inferences designed to support the validity of a theorem or hypothesis. A *proposition* is a symbolic representation of an expression or concept that evaluates to truth or falsity. A *theorem* is composed of propositions called *premises* whose intended purpose is to prove another proposition called the *conclusion*. Premises are facts known (or assumed) to be true.

Propositional calculus describes relationships between propositions using the logical operators AND, OR, NOT, IMPLICATION, and IF AND ONLY IF[1]. There are ten basic inference rules in propositional calculus, one introduction rule and one elimination rule for each logical operator [Ref. 122]. This simple set of rules is quite powerful and is sufficient for a system that deals with statements consisting only of propositional constants. For this reason, the ability of propositional calculus to represent the real world is limited to knowledge of a specific nature [Ref. 115]. A logical system supporting the concepts of variables (instances), predicates, functions, and quantification is needed for more general applications. This more general, and hence more powerful, system is called the predicate calculus.

2. Predicate Calculus

Predicate calculus uses the concepts and rules of propositional logic while giving added ability to represent knowledge in finer detail. This is accomplished through the introduction of predicates, objects, and quantifiers. In addition, predicate logic allows variables and functions of variables within a symbolic logic statement. Additional syntactic and semantic rules are also defined for the analysis of propositions, the interpretation of the resulting expressions, and the proving of valid deductions.

---

1. A variety of symbols and labels are associated with each logical operator. In this work, if reference to these operators is required, the capitalized label is used. Associated symbols in common use are $\wedge$ for AND, $\vee$ for OR, $\neg$ for NOT, $\rightarrow$ for IMPLICATION, and $\leftrightarrow$ for IF AND ONLY IF.

In predicate calculus, an expression is divided into two parts: the predicate and its arguments. Predicates represent what is known about the world and can assert some condition about or relating to the arguments[2]. Predicates may be used to indicate an argument's type or to denote a particular relationship between two or more objects. An example of a type predicate expression is *boy(alex)*, indicating that Alex is an instance of the type "boy". The expression *is_father(ron, alex)* contains a relationship predicate and represents the fact that Ron is Alex's father.

Arguments may be variables, in which case one of two special quantification operators are used to bind and delimit the scope of the variables. The standard quantifiers used in predicate calculus are the *universal* ($\forall$), meaning "for all", and the *existential* ($\exists$), meaning "for some". Rules relating to the addition and elimination of the two quantifiers, added to the ten inference rules of propositional calculus, constitute the complete set of rules for predicate logic. A thorough discussion of each rule is presented in [Ref. 122].

3. Reasoning

Logic is more than simply a system for expressing facts and knowledge in symbolic form. Rules of inference are used to derive new knowledge from old, or to prove the validity of some goal. The process of drawing inferences, either in the form of new facts or proofs, from premises is the basis for reasoning. Before discussing the automation of this process, it will be beneficial to explain the two types of reasoning: *deductive*, where general premises are used to verify a specific conclusion; and *inductive*, where established facts are used to draw a general conclusion. It is more common, at least within the field of computer science, to use the terms *goal-directed* and *data-driven* reasoning for deduction and induction, respectively [Ref. 46]. The terms forward and backward reasoning are also in general use; however, they are too easily confused with the rule-search control structures of for-

2. Because predicates and their arguments are merely used as a symbolic representation of an abstract concept, one cannot speak of "incorrect" predicate or argument names. However, some predicate names are certainly more descriptive than others. The selection of these names is a matter of good programming style.

ward and backward chaining found in production systems. Therefore, in this dissertation, explicit directionality will be applied only to chaining.

a. Data-driven Reasoning

This type of reasoning begins with a set of premises known to be true and applies the rules of logic to generate new facts. Specifically, reliance is on the inference rule *modus ponens* [Ref. 123], the elimination rule for the IMPLICATION operator. This rule states that given an implication and its antecedent, its consequent can be inferred; that is, if $R \rightarrow S$ is given and R is known to be true, S logically follows. This new fact may in turn be used in subsequent inference steps to produce yet more new facts, and so on. Typically, this type of reasoning attempts to generate data in support of some goal or theorem [Ref. 124]. Of course, data-driven reasoning may also be used simply to generate new knowledge without regard to a goal, perhaps as a method of learning. In any event, this inference mechanism relies on the existence of facts and works toward the satisfaction of the theorem. Hence, this process is also known by the terms *bottom-up*, *event-driven*, and *forward* reasoning.

Data-driven reasoning must contend with two problems. The first is the potential for the creation of inferences having no relation to the goal. The other drawback involves the difficulty identifying the shortest path of inference to the goal. Often, reliance is placed on a heuristic to guide the selection of premises from among those available. These problems are exacerbated if the number of available facts becomes large.

b. Goal-directed Reasoning

Another approach to reasoning is to start from a goal to be proved and to reason backward toward the factual evidence needed to validate the goal. This approach to reasoning is called *goal-directed*, also known as *top-down, goal-driven, or backward* reasoning. The goal is first decomposed into simpler, constituent subgoals. Each subgoal is then itself decomposed. The process continues until subgoals are attained that are a direct consequence of the premises. This process is characteristic of a reliance on the derived infer-

ence rule *modus tollens* [Ref. 123]. Modus tollens states that given the axioms R $\rightarrow$ S and $\neg$S, then $\neg\neg$R logically follows. This rule is derived in that it cannot prove anything not provable by the ten basic rules alone; however, it can help to simplify proofs by providing "short cuts" to the solution.

Most applications requiring the use of autonomous vehicles will be solved using goal-directed reasoning. The "divide and conquer" approach to problem solving seems best suited for human understanding in this domain [Ref. 18]. In addition, the act of decomposing the goal into simpler and simpler subgoals sheds light on which intermediate results must be attained and the sequence in which they must be attained. Supplied with this knowledge, a human user of an autonomous vehicle will possess a higher degree of confidence in its reasoning abilities because of this greater understanding of the problem.

C. AUTOMATED REASONING AND RESOLUTION

Despite the concise and unambiguous nature of predicate logic, its syntactic variety is not conducive to execution on a computer. As will be seen, automatic reasoners, for the sake of efficiency and in order to simplify the inference procedure, rely on a single rule of inference called the *resolution principle* [Ref. 125]. The specifics of resolution and its computer implementation are presented in this section.

1. Resolution

Resolution is an inference technique that solves logic problems by resolving (deriving) new knowledge, called *resolvents*, from known premises. The power of resolution, and its importance for the automation of problem solvers, lies with the fact that it subsumes the fundamental inference rules of modus ponens, modus tollens, chaining, merge, and reductio. Use of this principle requires that logical expressions be placed in the Conjunctive Normal Form (CNF). An expression in CNF uses only the OR and NOT logical operators. The purpose of this transformation is to provide for maximum uniformity and standardization in the representation of premises while eliminating the need for many of the connectives and quantifiers used in the propositional and predicate calculi.

There are six basic steps to converting predicate calculus expressions to CNF [Ref. 126]. First, the operators IMPLIES and IF AND ONLY IF are eliminated by replacing them with their logical equivalent. Second, the scope of negation is reduced to the atomic term level using DeMorgan's laws. Third, repeated variables are standardized based on the scope of their quantifiers. Fourth, existential quantifiers are eliminated by introducing constants and Skölem functions. Fifth, intermediate forms are converted to prenex form, in which universal quantifiers are eliminated. Sixth, conjunctives (AND operators) are removed along with extraneous parentheses. The premises in the final form consist of atomic formulas called *literals*. It is upon these literals that resolution operates.

Resolution works as follows. If x and y are two premises transformed into CNF, and i and j are literals of the premises x and y, respectively, and $i = \neg j$, then a new premise z can be derived by forming the union of x and y minus the literals i and j. The premises x and y are said to "clash" on the pair of complementary literals i and j. Expressed another way, the clause z is the resolvent of the parent clauses x and y. When dealing with complementary literals containing variables, an additional step is required prior to the application of resolution. During this step, the variables are matched and bound using the *unification* algorithm [Ref. 126][Ref. 127].

Other representative forms, such as Clausal form and Horn clause form, are even more restrictive but can improve the simplicity and efficiency of automatic reasoners. Clausal form is similar to CNF, except that the negated literals and non-negated literals of each disjunction are grouped together. If the negated group are placed on the right of an implication arrow, and the non-negated group on the left, the negation symbols can be dropped. The resulting clausal form is equivalent to but easier to interpret than CNF, at least for humans. So, for example, the CNF statement

$$\neg p \vee r \vee s \vee \neg q$$

becomes, in clausal form,

$$r, s \leftarrow p, q$$

with the interpretation "r or s is true if p and q are true". If one and only one literal is allowed on the left side of the arrow, the expression is said to be in Horn clause form. This is sig-

nificant because the syntax of Prolog rules is precisely that of Horn clauses. Hence, the Horn clause

$$r \leftarrow p, q$$

is written in Prolog as

```
r :- p, q.
```

and read "r is true if p and q are both true". Sets of these rules can be built which, when operated upon by resolution, emulate reasoning. This emulation, or more precisely implementation, of reasoning is known as chaining and is characteristic of rule-based systems. Specifics of these systems are given following a discussion of resolution and its relationship to the two types of reasoning.

2. Resolution and Reasoning

The basic resolution mechanism allows the derivation of new premises from old. Beginning with a set of initial premises and a theorem to be proved, resolution produces a resolvent from two of the original premises. This new premise may be combined with another premise to create another resolvent which is also added to the set. The proof succeeds when a resolvent matching the conclusion emerges. This pattern of reasoning moves in a forward direction from initial premises to concluding goal and is thus data-driven. As such, resolution is susceptible to the creation of inferences which have nothing to do with the proof at hand but whose spurious value cannot be immediately detected. In addition, there may be several possible paths of reasoning to the goal depending on the order in which the premises are combined. Circular reasoning is a trap that must be avoided as well.

Another approach to theorem proving is to start from the conclusion to be proved and reason toward the evidence needed to support the theorem. This pattern of reasoning is referred to as goal-directed reasoning and, with respect to resolution, is equivalent to proof by contradiction. First, the theorem (or desired goal) is negated and added to the initial set of premises. Assuming the set was consistent (non-contradictory) to begin with, the new set is now inconsistent. Resolution is then applied in the usual way. There are three possible

scenarios at this juncture: no premise exists containing the (non-negated) goal, in which the proof fails; a premise consisting only of the goal exists, in which case the proof succeeds because the goal and the negated goal resolve to a contradiction; or a premise containing the goal along with other literals is resolved with the negated goal to produce a new premise upon which resolution must be reapplied. In the last case, the literals of the resolvent represent subgoals that need to be resolved away if the theorem is to be proven. This method of theorem proving is called *resolution refutation*. One direct benefit of this approach is that irrelevant inferences are avoided, since only resolvents related to the negated goal are generated.

3. Resolution Strategies

Regardless of the type of resolution used, the drawbacks associated with both data-driven and goal-directed reasoning must still be addressed. The problem of combinatorial explosion of resolvents can be partly ameliorated through the choice of a resolution strategy. Three common strategies are breadth-first, set-of-support, and linear input [Ref. 46]. Each strategy guides the resolution process based on some heuristic. Although each have distinct advantages and disadvantages, the linear input form is of particular interest in this discussion because it is the strategy used by Prolog. In effect, linear input requires that every resolvent have a parent in the *base set*, the original set of premises. This strategy is simple, efficient, and, if applied to premises in Horn clause form, complete[3] [Ref. 115].

D. RULE-BASED SYSTEMS

Despite the attraction of basing knowledge representation on the formalisms of propositional and predicate logic, neither has provided significant insight into the machinations of intelligent behavior [Ref. 115]. It should be remembered, however, that these calculi were designed for other purposes. This shortcoming has led to the development of rule-based production systems for the emulation of intelligent behavior. Humans tend to asso-

3. Completeness is the property whereby an inference mechanism is capable of generating all valid conclusions that can be drawn from a set of premises.

ciate intelligence with regularities in behavior [Ref. 128], and the expression of these regularities (at least as they are understood by the AI community) lends itself well to rules [Ref. 129]. Thus, in expert systems, production rules generate behavior much the same way that functions in Lisp and relations in Prolog do. However, *production systems* (systems composed of production rules) are designed to be employed in a procedural fashion, the intent being to manipulate the current state of the problem towards a state closer to the solution [Ref. 124]. To accomplish this, the system is said to drive production rules in a forward or backward direction. This process of control is called *chaining*.

1. Production Rules and Systems

Production rules were originally used in theorem provers called *canonical systems* [Ref. 130]. A canonical system contains an alphabet for making strings, some axiomatic strings, and a set of productions. Each production is of the form

$$a_1\Sigma_1...a_m\Sigma_m \rightarrow b_1\Sigma'_1...b_n\Sigma'_n$$

where each a_i and b_i is a fixed string; a_1 and a_m may be null; some or all of the a_i and b_i may be null; each Σ_i is a variable string which can be null; each Σ_i is replaced by a certain Σ'_i; and the arrow signifies a rewrite function. The expression $a_1\Sigma_1...a_m\Sigma_m$ is called the *antecedent* of the rule and $b_1\Sigma'_1...b_n\Sigma'_n$ the *consequent*, just as in the conditional statement of propositional calculus. The arrow, however, does not correspond to the logical IMPLIES relationship. This simple system, with the ability to scan an input string of symbols and perform addition or deletion of symbols, is all that is required to verify proofs in a formal system.

In expert systems, the focus is on the solving of problems. Production rules in these systems differ little from the rewrite rules of canonical systems, except that with expert systems the key is the transformation of some initial state to one which satisfies a criterion representing the solution of the problem, rather than the generation and recognition of symbolic structures. To this end, expert systems like MYCIN [Ref. 131] and systems built with expert-system shell languages such as OPS5 and CLIPS have been developed. In

these systems, the alphabet and string-based axioms of the canonical system are replaced with structures built of objects, attributes, and values. Taken together, this database of structures makes up the *working memory* of the system and generally includes the premise facts, goal conditions, and intermediate results that comprise the state of the problem. To enhance the efficiency of a production system, the working memory is normally stored in a high-speed memory. The time of creation or the most recent use of each data is considered by the system when determining the placement of data in the high-speed memory. This strategy is called *caching* [Ref. 123].

A rule of a production system is a two-part statement that embodies knowledge. Each rule consists of an antecedent-consequent (or condition-action) pair. The antecedent of the rule, which consists of one or more premises, is matched with the various symbol structures known to the system using various search and pattern-matching techniques. In addition, the process of matching may involve the binding of variables. If a successful match is made (i.e., the premise(s) of some rule are true), then the rule is said to be *active*. Subsequently, the active rule is fired, and the actions (or conclusions) specified in the consequent of the rule are performed. Hence, production rules have the syntax

$$P_1, \ldots, P_m \rightarrow Q_1, \ldots, Q_n$$

where each P_i is a premise, each Q_i is an action, and if all the premises are matched to the contents of working memory and the rule is selected for firing, then all actions are performed. Another interpretation is possible, in which each P_i is a condition, each Q_i is a conclusion, and if all conditions are satisfied, the conclusions may be drawn. In this instance, rules are generally expressed with the antecedents on the right hand side, the conclusions on the left hand side, and the arrow pointing to the left. In either case, the set of production rules make up the *knowledge base* of the rule-based system.

A production system, besides consisting of a knowledge base and a working memory, must also have a mechanism for the selection of rules and the subsequent application of the rule consequent. This mechanism is known as the *rule interpreter* or *inference*

engine. The inference engine is itself a software system that manages the search and pattern-matching operations associated with the rule set. During an execution cycle, the inference engine examines the working memory in an attempt to activate one or more rules in the knowledge base. A rule is then selected by the interpreter from the list of active rules and the appropriate actions carried out. Often, as a side effect of this process, the contents of working memory are manipulated, reflecting the new state of the problem. If more than one rule is placed on the activation list, the interpreter must decide which one to fire. This process is known as *conflict resolution* and is typically based on some simple heuristic. Such heuristics guide the interpreter and are usually based on recentness of rule activation, recentness of rule firing, the specificity of the antecedents, or, more simply, priority.

If all conflicts encountered by the system are resolved in a sequence specified by the designers of the system, the rule set is said to be *deterministic*. This characteristic of expert systems is very important when applied to the control of autonomous vehicles. A simple strategy for insuring determinism is the requirement that, for all working memory configurations, only one rule is ever eligible for activation. In some implementations, this constraint may prove to be too restrictive; therefore, the assignment of priorities to rules involved in potential conflicts will also ensure a deterministic ordering.

The basic match-select-apply cycle constitutes the problem-solving process used by the inference engine of a rule-based system to reason and draw conclusions about a particular problem. The cycle continues until either a fired rule contains an explicit command to halt or an empty activation list is encountered. Generally, an explicit halt command will arise because a goal state has been achieved, even though the goal represents an anticipated state of failure. An example of this would be the "trap" state entered because the power available to the robot is insufficient to continue the mission.

2. Chaining

Separate from the issue of conflict resolution is the direction in which the various rules are employed by the inference mechanism. This direction of rule *chaining* may be for-

ward, from conditions known to be true towards problem states established by those conditions; or, the direction may be backward, from some goal state towards the conditions supporting its establishment. This is quite similar to the forward and backward strategies found in resolution problem solving discussed earlier. Because chaining is usually constrained to one direction, forward and backward chaining are both considered to be special cases of resolution [Ref. 123].

In a forward chaining expert system, the production rule antecedents are found on the left hand side of the rule. This part of the rule is also called the *conditional* or the IF part of the rule. The forward chaining inference engine attempts to activate rules by matching the conditionals of the rules against working memory. Once the conditional part of a rule has been matched, it can be fired. In some cases, more than one rule conditional is satisfied by the contents of working memory. These rules are activated and one is fired according to some selection criteria applied by the inference engine. When a rule is fired, the actions specified by the rule are carried out. These actions are listed in the rule's right hand side, also called the *consequent* or the THEN part of the rule. If the executed consequent alters the contents of working memory, a new cycle of searching, matching, and rule firing may commence.

A backward chaining inference engine starts the search for a solution with the conclusion (or goal). The data base is examined for the goal. If it is there, the search stops and the goal is proven. Otherwise, the inference engine turns its attention to the rule base. In the backward-chaining system, the antecedent is on the right hand side of the rule; thus, if the goal represented by the left hand side is to be established, the subgoals specified on the right hand side must first be satisfied. Subgoals of the antecedent may be related by one or more logical connectives. The subgoals are satisfied if they can be matched to facts in the database or if a rule consisting of the subgoal on the left hand side can be satisfied. The corresponding left side of matched rules are used as intermediate hypotheses which are maintained by the inference engine. Each conclusion is immediately applied against the rule base, thereby building a search chain. Again, search continues until the original hy-

pothesis is proved or until all rule-matching possibilities have been exhausted. It is possible that no match between an intermediate hypothesis and rule right sides is possible. In this case, this search path is terminated, and the search process selects an untried search path. An attempt is then made to match this subgoal to the right side of some different rule. If no further matching can be done and all possible paths have been explored for the satisfaction of the goal, the search fails and the inference cycle stops.

The distinction between how forward and backward chaining differ can best be shown through an example[4]. Given the following rule set,

(R1) $X \Leftrightarrow aXa$

(R2) $X \Leftrightarrow bXb$

(R3) $X \Leftrightarrow cXc$

and a start symbol *p*, palindromes can be either generated using forward chaining, indicated by the right-pointing direction of the arrow, or recognized using backward chaining, indicated by the left-pointing direction of the arrow. If the forward-chaining rules are applied in the order R1, R2, R3 to the start symbol, the system will generate the strings *apa*, *bapab*, *cbapabc*, respectively. This is an example of forward chaining because the symbol *p* and all newly generated strings are matched against the antecedent (left-hand side) of a rule and a conclusion detached from the instantiated right-hand side.

The same rule set can be used to recognize (or "prove") palindromes. If a target string *cbapabc* is given, the sequence of backward-chaining rule applications leading to the construction of this palindrome can be traced. Matching the right-hand side of R3 to the target string results in *bapab* from the instantiated left-hand side of R3. The chain continues by matching the right-hand sides of R2 and R1 in order, yielding the start symbol *p*. Because this represents an acceptable condition for the existence of the original string, the palindrome is recognized. A target string such as *abcabc* will not satisfy the right-hand side of any rule, and therefore cannot be recognized as a palindrome within the scope of this

4. This example is derived from one used in [Ref. 115].

knowledge base. Note that this system would not be capable of recognizing the string *dpd* even though it is a palindrome. Nevertheless, production systems are usually not intended to be complete in every aspect of knowledge in a specific problem domain [Ref. 124].

In the theorem-proving literature, forward chaining is usually associated with "bottom-up" reasoning while backward chaining is typically associated with "top-down" reasoning [Ref. 46]. The type of reasoning and the direction of chaining are not the same, however, and the direction of reasoning does not necessarily imply the direction of the chaining used. The chaining implements the reasoning, and the reasoning controls the chaining. Newell has noted [Ref. 132] that chaining is a phenomenon of the symbol level, where the emphasis is on right-hand sides and left-hand sides of rules, whereas reasoning occurs at the knowledge level, where the distinction can be made between facts and tasks.

Hence, the choice between a forward and backward chaining expert system for the high-level control of autonomous vehicles depends primarily on the shape of the search space of the problem. The search space results from the reasoning process. If the problem contains substantially more facts than conclusions (goals), backward chaining will produce the shortest path in the least time. If, on the other hand, the problem consists of a few facts and potentially many (or unknown) goal states, forward chaining will typically yield the quickest solution [Ref. 133].

As the number of facts and rules grows, the forward chaining system faces the increasingly difficult and time consuming task of determining the applicability and order of rule firing. Unlike backward chaining systems, where the goal-directed nature of the inference engine guides the execution of the rules, the forward chainer must continually compare every fact in the current working memory with the antecedent of every rule. Before long, the number of comparisons required to complete this becomes unmanageable. At this point, a heuristic-based search algorithm is typically introduced to improve efficiency [Ref. 124].

Other considerations besides efficiency may require that a forward chaining system be applied to a problem well suited for solution by backward chaining. By placing an

explicit ordering on the goals to be achieved by the system, it is certainly possible for a forward-chaining expert system to implement goal-directed reasoning. The two systems, working on the same problem and operating under the same conditions, will formulate the same results [Ref. 134]. Besides differences in efficiency, however, the systems may also exhibit different sequences of intermediate results. This topic is addressed in the next section of this chapter.

E. MISSION REPRESENTATION

The purpose of the rule-based production systems discussed above is to solve problems. It is beneficial at this point to focus the problem domain and to identify a particular kind of production system applicable to the control of autonomous vehicles. An autonomous vehicle must have the capacity to act intelligently. In most cases, this intelligence will be supplied by a human intimately familiar with the goals and subgoals required of the vehicle. This knowledge will be codified in the form of a rule-based expert system. Therefore, the purpose of the expert system is to provide the direction, experience, and insight in support of mission accomplishment that would otherwise be forthcoming from a human operator. The applicable problem domain, then, is the set of potential missions as defined by the user of a particular vehicle.

To represent a mission is to describe, to the best ability of the mission specialist, what is required of the AV. Information such as the detailed specification of goals, their decomposition, the sequencing of subgoal achievement, measures of success, and alternative acceptable solutions may or may not be included. In any case, it is essential for the safety and security of the vehicle (and of personnel and property relying on the vehicle) that this description of the mission be in an unambiguous and orderly form. Rule-based systems are available for the implementation of such problems on a computer. However, the relationship between the rules is not readily evident through inspection of the rule set. As an aid to the design, verification, and validation of these systems, graphical representations have been devised [Ref. 135]. Two possible approaches to representing rules and their relation-

ships are introduced here: the AND/OR goal tree and the State Transition Diagram (STD). These structures are not new and have been used to varying degrees in the fields of digital design, control systems, robotics, and software engineering [Ref. 121][Ref. 136][Ref. 137]. To meet the requirement for unambiguous execution of missions depicted by these structures, additional information constraining their traversal is required. Nevertheless, both support the representation of a search space associated with a set of production rules, both explicitly account for the interconnections between the rules, and both are laid out in a style compatible to a particular direction of chaining. Hence, they are ideal for the purpose of mission representation.

The topic of search, as it relates to rule-based systems, is discussed next, followed by a detailed description of the AND/OR goal tree and the State Transition Diagram. In applications involving autonomous vehicle control, equivalent representations are achieved only if the sequence of side effects resulting from the search is the same.

1. Search

Given a representation of a problem in the form of a graph structure, the method of *search* must be considered. A road map is a simple example of such a graph. The goal of the search may be to identify the shortest route between two points assuming that travel must be constrained to the existing roads. Or, the search may involve the inclusion of a set of specified waypoints. In the domain of mission execution, the search will provide a time-ordered sequence of subgoals, tasks, or states which lead to the attainment of a specified goal state. To ensure that this sequence is the one that is expected, the search must adhere to a specific set of constraints.

There are two basic categories of search: blind and heuristic. Blind search is orderly and methodical and will eventually solve the problem if a solution exists. This simplest form of search utilizes a trial-and-error approach for solving a problem by examining all alternatives provided by the knowledge base. This process is called *generate and test*; that is, a possible solution is generated and then tested to see if it satisfies the goal condi-

tions. As search spaces become large, the search mechanism faces the phenomenon of combinatorial explosion, in which new nodes are added to the search structure exponentially. In these circumstance, blind search may consume an unacceptable quantity of computational or temporal resources. For this reason, heuristics can be added to blind search to provide guidance beyond the exhaustive nature of that approach.[Ref. 46]

A heuristic is formulated based on experience and empirical knowledge to guide the search by narrowing the search process. This is accomplished by eliminating infeasible or unpromising search paths from the set of potential solutions. The use of heuristics does not guarantee that a better solution will be found, or that a solution will be found any faster than that found by blind search. Often, a heuristic must be tailored to a particular problem; even then, it may be difficult to predict the impact on search speed and efficiency.

Typically, a heuristic involves a mathematical function that results in a numeric value [Ref. 133]. This value can then be compared against other heuristic values or numbers representing constraints on the solution. Since the operation that takes the search from one node to the next has an associated cost, the mathematical functions are also called *cost functions*. Some heuristic techniques are popular due to their general applicability. Examples include hill climbing, best-first, and A* search [Ref. 138]. With respect to the inference cycle of an expert system, search techniques are employed during the pattern-matching phase, when rules are being identified for activation. The particular search strategy and the type of knowledge representation used in an expert system are not independent of each other; often, during the design of a system, the selection of each must be done in concert [Ref. 115].

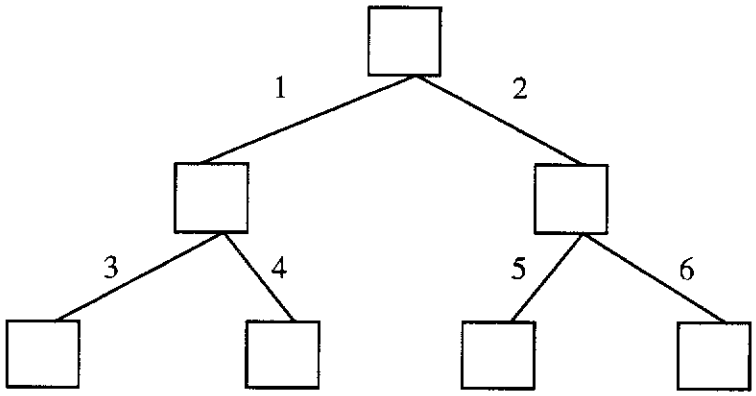
Blind search is still a viable approach, however, particularly when the problem has a small to medium sized search space. Moreover, as computer speed continues to increase, such brute-force search techniques become more viable. Two popular blind search methods are Breadth-First search (BFS) and Depth-First search (DFS). These differ primarily in the order in which the nodes of the search structure are visited. In BFS, the root node is searched, followed by an examination of all the root node's children. When this has been

completed, the nodes in the next level are searched, and so on until the goal node is reached or until all nodes have been searched (Figure 4a). A characteristic of BFS is that the shortest path between the root and the goal will always be found [Ref. 46]. Typically, the search proceeds downward from top-to-bottom and left-to-right. Similarly, DFS begins at the root and proceeds through the tree from a top-to-bottom, left-to-right fashion; however, the search progresses along a branch as opposed to a level of the tree (Figure 4b). If a goal state is not reached on the current branch, the process returns to the last node from which an untried path is available. As in BFS, the search continues until the goal is reached or until all paths are traversed. A problem with DFS is that the search may spend much effort in a deep subtree, far from a goal which may exist in at much higher level. An advantage of DFS is that it is implemented simply and efficiently.

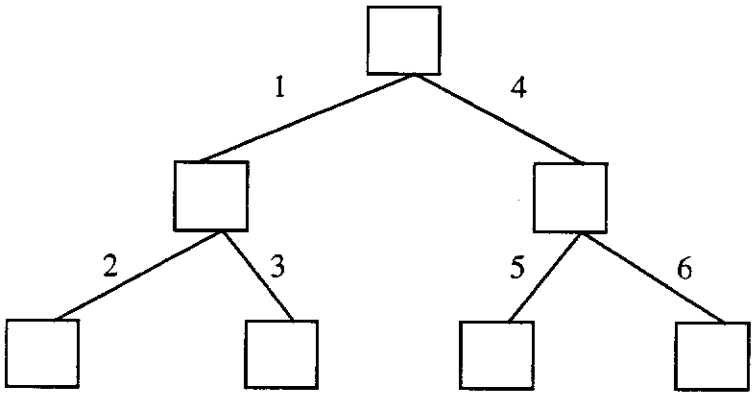
The search spaces associated with the fundamental, high level control of autonomous vehicle missions are typically not large enough to benefit from heuristic search. Furthermore, the necessity for determinism on the part of the inference mechanism supports the use of a predictable, exhaustive search technique like DFS or BFS. On the other hand, path-planning and replanning algorithms which may involve much larger search spaces and less restrictive timing requirements would be good candidates for heuristic search.

2. AND/OR Graph

State graphs, by definition, may contain cycles. This may properly represent the problem at hand; however, infinite loops are not computationally feasible. For this reason, state graphs are often converted to a cycleless structure called a *search tree*. The initial state node is called the root of the tree; the terminal nodes are the "leaves" of the tree, those nodes with no children; and all remaining nodes make up the intermediate nodes (subgoals) of the tree. Each intermediate node has exactly one parent and at least one child. The root node comprises level 0 of the tree; the children of the root comprise level 1; the set of children of level 1 nodes make up level 2; and so on. The nodes of the tree are connected to each other by arcs.



(a) Breadth-First Search



(b) Depth-First Search

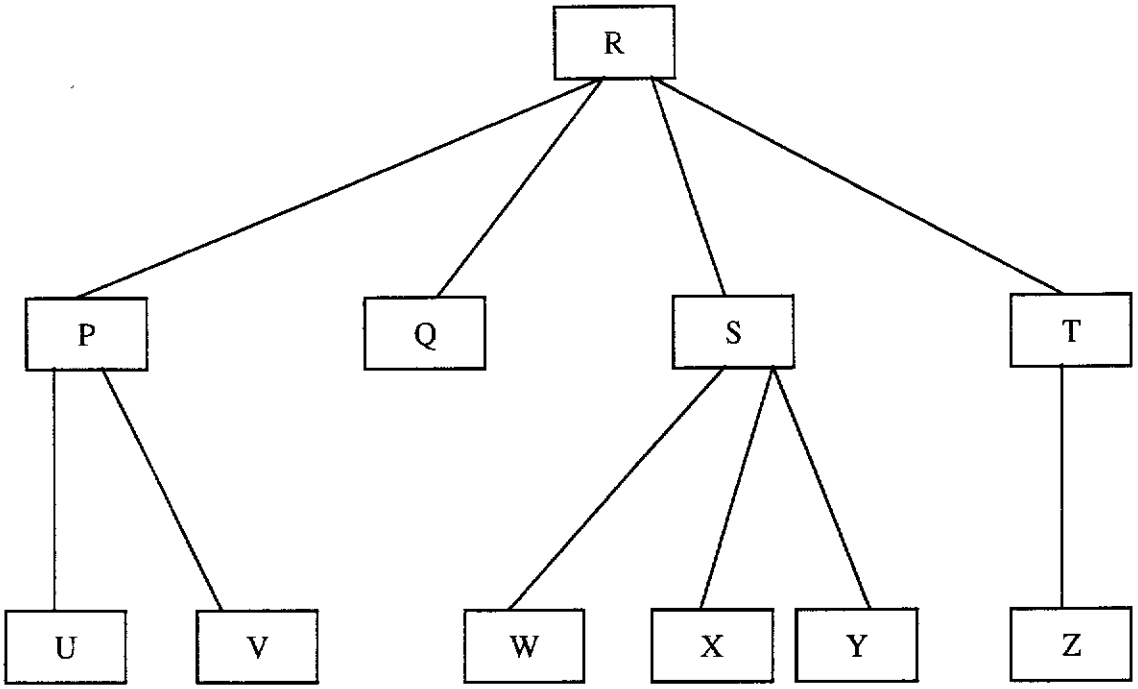
Figure 4. Blind Search Strategies

A further refinement to the search tree is the AND/OR tree [Ref. 46][Ref. 139]. In this structure, the branches from a node to its successors may be related in one of two ways. Branches representing alternative paths have a logical OR relationship; that is, one branch of several may be the path to the goal. In other circumstances, the achievement or traversal of multiple subgoals may be required before the goal is satisfied. These subgoals are related through a logical AND operator. In this case, each subgoal must be satisfied or the problem cannot be solved. A combination of logical AND and OR relationships may be represented by the AND/OR tree as shown in Figure 5. In this figure, the arcs leading to the subgoals related by an AND share a single point on the lower edge of the parent node. This grouping of arcs is referred to as a *hyperarc* [Ref. 137]. Single arcs leading away from a node identify a subgoal related to the parent's other subgoals through an OR relationship. Separate hyperarcs associated with the same parent node are logical disjunctions.

These structures are static in that they only convey the logical relationships between nodes but nothing about the sequence in which the branches are searched. However, this information is essential to the validation of goal specification and attainment by autonomous vehicles. In the particular case of the Rational Behavior Model, goal sequencing is explicit and hence must be reflected in the graphical structure. For this reason, AND/OR goal trees are used to express goal decomposition as well as execution sequence. Specifically, a depth-first (top-down, left-to-right) traversal, as used by Prolog, is assumed.

3. State Transition Graph

The conventional State Transition Diagram (STD) [Ref. 136] is a modeling tool for describing the time-dependent behavior of a system. Each node of a state transition graph represents one of the possible states of the problem. A state is a set of attributes characterizing a system at a given time such that no knowledge of past inputs is needed to determine the future behavior of the system. Generally, the number of states in the system's set of possible states is finite. The directed arcs in an STD represent valid state transitions which occur when a specific condition is detected. A condition is an event in the environ-



$R \leftarrow PQ + ST$
$P \leftarrow U + V$
$S \leftarrow W + XY$
$T \leftarrow Z$

Figure 5. Representation of a Rule Set as an AND/OR Tree

ment of the system, such as an interrupt or the arrival of a data packet, that typically leads to one or more actions taken by the system. The arc points to the state entered after the actions have been applied. In a multilevel STD, any individual state of a higher-level diagram can encapsulate a lower-level diagram that further describes the higher-level state. The final state(s) in the lower-level diagram then corresponds to the exit conditions in the higher-level state. This decomposition, which may be recursively applied, gives order and understandability to an otherwise complex diagram and represents a standard approach to the design and analysis of complex, state-based systems [Ref. 136].

The notation used for conventional STDs requires that responses, in the form of state transitions, be identified for all possible conditions. This may lead to states which have two or more successor states. To avoid nondeterminism, the conditions for each transition path must be exhaustive and mutually exclusive. For example, if some condition X is needed to trigger a transition from state A to state B, and some other condition Y causes a transition from state A to state C, then it must be true that $X \cap Y = \varnothing$.

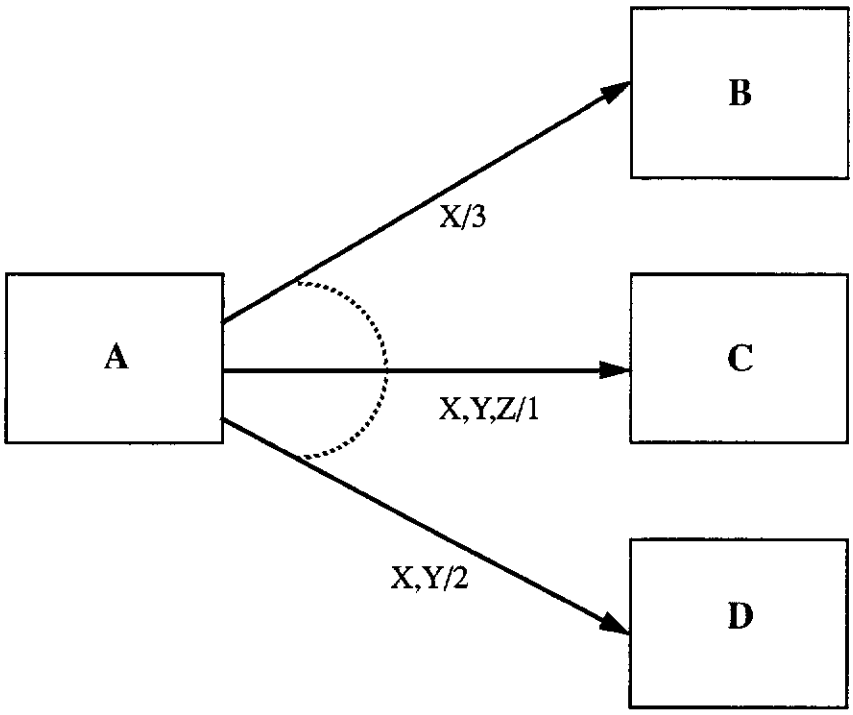
A state transition diagram describing a mission based on the forward-chaining control approach may involve states which have multiple successor states but transition conditions which are not mutually exclusive. Following the example from the previous paragraph, this would result when, for conditions X and Y, $X \cap Y \neq \varnothing$. This gives rise to a conflict, the resolution of which must be accomplished to ensure the proper (i.e., expected) move is made. Since traditional STD notation does not make allowance for this, an extension called the state transition diagram with path priorities is now introduced. This graphical representation is similar in purpose to the Logical Path Graphs of [Ref. 135], except that nodes refer to individual search states, and explicit priorities are assigned to transitions which may be in conflict with each other. This would occur when the associated rules were activated at the same time [Ref. 140].

DEFINITION: A *State Transition Diagram with Path Priorities* models the time-dependent behavior of a system containing state transition conditions which may not be mu-

tually exclusive. Conflicts in determining the next state are resolved through the use of pre-assigned, numerical *path priorities.*

A state transition diagram with path priorities is shown in Figure 6. This diagram reflects essentially the same information as a standard STD. Again, each node represents a valid state of the system. An edge from node i to node j indicates that, as a result of rule i firing, rule j will be activated. Note that such an edge does not necessarily mean that rule j will fire immediately after rule i, only that rule j will be added to the *agenda* (the list of active rules). Conflicting state transitions, in the form of simultaneous activations caused by conditions coincident with each other, are represented by multiple arrows extending from one state and joined by a broken arc. This indicates that one, all, or some combination of the transition paths may be active depending on the current conditions in which the system finds itself. The child nodes pointed to by these arrows represent the possible successor states, and the path selected depends on the conflict resolution strategy employed by the inference engine. The strategy employed is determined *a priori* by the designers of the expert system. In the example of Figure 6, numerical priorities and alphabetic conditions are associated with each conflicting transition path. If a conflict should arise, as in the case when conditions X, Y, and Z are all applicable, the arrow associated with the highest-priority active transition is the path taken. The use of priorities is consistent with the *depth* resolution strategy, as, for example, in the left-to-right order in which Prolog treats disjunctive clauses [Ref. 141]. It should be noted that the STD of Figure 6 may execute as part of a high-level, mission control loop. Therefore, the combination of active successor states to state A may change from iteration to iteration.

Both AND/OR goal trees and State Transition Diagrams are useful for the visualization of the relationships between rules in a rule set. However, it should not be inferred that either is essential in the design or implementation of such systems. In practice, these structures are not commonly employed for any but the most concise problems, as they simply become too unwieldy. Instead, software tools specifically designed to assist in expert system creation and maintenance should be utilized to manage complexity [Ref. 142].



COND XYZ	ACTIVE TRANSITION	EXECUTED TRANSITION
000	-	-
001	-	-
010	-	-
011	-	-
100	AB	AB
101	AB	AB
110	AB,AD	AD
111	AB,AC,AD	AC

Figure 6. State Transition Diagram with Path Priorities

F. SUMMARY

This chapter introduces the concepts of automated reasoning pertinent to the Rational Behavior Model software architecture, a concept presented in the next chapter. The solution of problems, in the form of mission planning and control, must be achieved by the human expert prior to developing the expert system capable of controlling the autonomous vehicle in hostile environments. Missions are developed through the application of a reasoning process. Typically, in the domain of autonomous vehicle mission control, goal-driven reasoning is required since the explicit sequencing of subgoals and their execution is involved. Next is the translation of the resulting logical steps into a form that can be readily manipulated by a computer. Rule-based languages have been developed for this very purpose. Rules are created which embody the logical relationships between what is known and what is to be achieved. The rules are then gathered together along with certain facts to form a knowledge base. An inference engine, by searching the knowledge base, can then infer new knowledge from existing facts. If the knowledge base has been designed to emulate the reasoning performed by the human operator of a vehicle under a variety of circumstances, the resulting expert system may reasonably be expected to successfully control a like vehicle autonomously. As will be developed further in the next chapter of this dissertation, the concept of using an expert system to exercise mission-level control over an unmanned vehicle is fundamental to the Strategic level of the Rational Behavior Model.

V. THE RATIONAL BEHAVIOR MODEL

The Rational Behavior Model (RBM), a tri-level software architecture for the control of Autonomous Vehicles, is defined in this chapter. Each level is explained in terms of its functionality, abstraction mechanisms, implementation restrictions, and its interface with the entity to which it is joined. This discussion is preceded by an introduction to several related tri-level control software architectures and a discussion of fundamental concepts relevant to this work.

A. INTRODUCTION

Traditional approaches to the development of software for autonomous vehicle control systems have typically involved significant numbers of individuals working independently but communicating frequently so as to insure consistent interfaces between software components [Ref. 143][Ref. 144]. Typically, these systems are constructed from requirements provided by end users who have well conceived ideas for what the system is to accomplish and the sequence of tasks required to attain these goals. These requirements are passed to software systems analysts whose job it is to design and implement software that realizes the user's requirements. If a physical system is involved, such as an autonomous vehicle, yet another group of specialists must be consulted to insure proper integration of the software with the hardware. Thus, construction of a software system for the overall control of an autonomous vehicle must involve the coordination of and the reliance upon individuals from varying backgrounds. In addition, software incorporating concepts at a high level of abstraction (mission planning and control) must interface with software concerned with a much lower level of vehicle control (stability and hardware manipulation). One solution is for the software analysts to ask these experts for their requirements which are subsequently translated into a form understandable by computer scientists. This is a time consuming process fraught with the potential for error, misunderstanding, and ambiguity. Faulty require-

ments inevitably result in the alteration of the system's design. Any partial realization of the original design will, of course, be affected.

Once a complex software system is delivered, it may be subject to requests from the user for additional or modified functionality. Software systems which control autonomous vehicles are especially susceptible to these requests, since such vehicles will be expected to perform a wide range of missions. New missions may imply new capabilities, affecting hardware, software, or both. Changes of this nature are often very difficult to realize in software because the "high-level" competency of the system is typically composed of many simpler capabilities inextricably intertwined throughout the system [Ref. 56][Ref. 55][Ref. 23].

These two problems, the effective linking of high-level symbolic computation with low-level vehicle control software and the modification of same, directly relate to the *architecture* of the software system. For a particular problem domain, a software architecture founded upon levels of abstraction will allow for expression of the problem in terms recognizable by the experts in the given problem domain (or subdomain) [Ref. 55].

DEFINITION: A *software architecture* of a system encompasses the conceptual design and organization for the software. The architecture includes, but is not limited to, a description of the abstraction mechanisms, division of responsibility, and specification of external interfaces through which the software entities communicate with each other and the external world. Each software entity, also called a *module*, is an independent conceptual unit within a software system.

The software architecture is an essential component in the design of a software system for the control of an autonomous vehicle, because therein lies the key to software maintenance, reuse, and modification. Most existing software architectures omit from their conceptual definition any reference to the programming paradigm best suited to implement that abstraction. This omission can only weaken the architecture, since selection of an inappropriate language may introduce inefficiency and negatively affect understandability.

By selecting an appropriate programming paradigm, the inherent power of the language can be brought to bear on the problem. Additionally, the domain expert, given the proper training, may become directly involved in the implementation of the system. The obvious advantage to this approach is that the expert can apply knowledge directly without having to verbalize it to a software analyst. This is particularly useful during the acquiring of knowledge for the construction of an expert system at the higher symbolic level and during the integration of the lower numeric-level control software with the rest of the system. Of course, there must exist an intermediate level providing an interface between the two disparate outer levels.

Abstraction mechanisms, in concert with the programming paradigm, can also accommodate requests for system modification. Areas of the system effected by changes in requirements may be isolated within a single abstraction level. Architectures that can accommodate change are characterized by constituent software modules which have well-designed external interfaces, strong cohesion, and minimal coupling [Ref. 145]. Cohesion defines how closely related the internal elements of a software module are to one another, and coupling is a measure of the strength of the interconnection among software modules.

The RBM architecture has been conceived with these features in mind. To accomplish this, each level of RBM embodies a particular aspect of the overall control problem. In addition, particular programming paradigms have been selected, based upon their expressive power, for the realization of these levels. Although dividing control software into multiple levels of abstraction is not a new concept [Ref. 37][Ref. 39], employing multiple programming paradigms in the implementation of those levels is much less common [Ref. 146]. The expression of global behavior through the use of a top-level, rule-based doctrine is unique to RBM and its antecedents [Ref. 23]. Each level of RBM is characterized by a specific approach to design and implementation which uniquely identifies this software architecture.

The Rational Behavior Model was first introduced in [Ref. 147]. The informal concept of RBM has since been refined with the results presented here. Included is the addition of several important, defining constraints placed on the original model. These constraints are

deemed necessary if the desired traits of software development and flexibility described above are to be implemented practically. In addition, this work provides definitions for many terms and concepts associated with the control software of autonomous mobile systems in general and RBM in particular.

B. TRI-LEVEL CONTROL SOFTWARE ARCHITECTURES

Multi-level organization of complex systems has been utilized for millennia in the form of military, government, and business bureaucracies. The success of these hierarchies is founded on the general decision-making that occurs at the top level and the increasing specialization of functionality at the bottom levels. Resources can be applied most efficiently when the overall problem at hand has been decomposed into a finite number of simpler, more easily understood, tasks. The degree to which problems are decomposed, and the number of levels into which the hierarchy is divided, is usually resolved by experience based on one or more abstraction criteria. This concept applies equally well to the software solution to the problem of mission and vehicle control of autonomous agents.

As the complexity and scope of real world applications involving autonomous agents increases, the single-level control architectures of the past have proven to be inadequate in dealing with the problem of software complexity [Ref. 37]. Hence, software engineers have utilized mechanisms based on abstraction to deal with this problem [Ref. 145]. This has led naturally to multi-level software hierarchies which exhibit similarities to human bureaucratic structures. The number of levels resulting from this process is somewhat arbitrary; however, a tri-level organization supports coherent abstraction while retaining simplicity of implementation and efficiency of communications. Some prominent examples of tri-level control software architectures and the important characteristics of each are described next.

1. CMU Mobile Robot

This system, which evolved directly from the Stanford Cart, was developed during the time frame from 1981 to 1984 [Ref. 17]. The concept of control used for this

robot was based on a three-level hierarchical structure. At the top was the "Planner" level, followed by an intermediate "Plan Execution" level, and a lower "Actuators and Sensors" level. The levels communicated with each other by posting messages to a *blackboard* [Ref. 66] data structure.

2. Saridis

An early proponent of "intelligent" control of robotic vehicles, Saridis recognized three basic levels of control called the *organizational*, *coordination*, and *hardware control* levels [Ref. 37]. According to Saridis, these levels fall within the research areas of artificial intelligence, operations research, and control theory, respectively. Further work by Saridis has provided a basis for the design of the middle, coordination level through the use of Petri Nets [Ref. 148]. However, an architectural instantiation utilizing these ideas has yet to appear. This work is significant due to its proposed conceptual organization of intelligent robotic system control software.

3. SSS

This architecture [Ref. 149] consists of three levels characterized by their treatment of time and space. SSS is an acronym for *Servo*, *Subsumption*, *Symbolic*, levels which imply a progressive quantization of first space and then time. The lower, servo level operates in a domain of continuous time and continuous space; that is, the state of the world is being constantly monitored by the servo system, and this state is typically represented in a numerical format. The middle subsumption level also continuously checks the state of the world, but generally responds only to certain specific situations. In this way, the state of the world is discretized into a small number of special task-dependent categories. Thus, this "subsumption" level is said to operate in continuous time and discrete space. The Symbolic level, while also operating as "situation recognizers", reacts only on the basis of significant events. Thus, both space and time are discrete at this level.

Each level of SSS is seen to operate individually, with inter-level coordination occurring over carefully designed interfaces. Each interface consists of a command link from

higher to lower and sensor link from lower to higher. Commands from the symbolic to the subsumption level act to parameterize certain modules. By turning behaviors on and off, the Symbolic level guides the system in its attainment of the goal by introducing cooperation amongst the possibly conflicting behaviors of the Subsumption level. The controllers of the Subsumption level adjust setpoints which are sent to the servo loops of the Servo level over the command interface between these two levels.

The sensory interface from Servo to Subsumption level carries unprocessed "environmental" data fed to a signal-processing front-end located in the Subsumption level. This device consists of various matched filters designed to equate certain classes of sensory states since they would all call for the same motor response by the robot. Other classes requiring a different response by the robot are discriminated by the matched filter recognizers. If a specified group of recognizers are all valid, this leads to an event detection signal being passed from the Subsumption to the Symbolic level for processing.

The SSS architecture has been implemented to solve navigation problems for a wheeled laboratory robot, as described in [Ref. 149]. High-level, "strategic" control provided by the Symbolic level is determined by referring to a geometric map of the robot's world. It is unknown whether this architecture could be adapted to handle a dynamic environment such as an autonomous vehicle might find itself. Nevertheless, the SSS architecture recognizes the importance not only of conceptual abstraction for the solution of the control problem, but also the requirement to design specialized interfaces between the levels. These important features are shared by the Rational Behavior Model.

4. Open Robot Controller

This work involves a three-level architecture for the control of robotic systems developed by the Institut de Recherche en Informatique et en Automatique (INRIA) of France. An application is created from an existing library of *robot-tasks*. Each robot-task associates a local behavior related to events recognized by sensory signals and a control scheme dedicated to some physical device [Ref. 150]. An object-oriented design approach

is taken with respect to the robot-tasks. By doing this, coherence between the robot-tasks is ensured, understandability is improved, and realization of a man-machine interface is facilitated. In addition, reasoning, expressible in terms of predicate logic, is available to the application designer through an as yet unnamed programming language [Ref. 58]. Furthermore, applications expressed in this language are subsequently translated into Esterel, a synchronous, parallel, imperative language especially well suited for the implementation of *reactive* systems [Ref. 111]. In this context, the term "reactive" equates to "event-driven" whereby the behaviors are initiated through the object-oriented message-passing mechanism. This architecture utilizes the abstraction mechanism of class hierarchies at the task (middle) level.

5. Events and Actions/ARCS

This control software architecture [Ref. 55] has been designed specifically to support the Autonomous Remotely Controlled Submersible (ARCS) built by International Submarine Engineering (ISE) of Canada. This system incorporates the concepts of object-oriented programming and techniques of event-driven, real-time software. Most importantly, an emphasis on mission-reconfiguration improves the ease and rapid customizing of software for different applications. The architecture is constructed from software objects, or components, which respond to external inputs (events) by producing appropriate outputs (actions). These components are gathered and maintained in a library. Construction of a real-time control software system is then a matter of selecting the required components from the library, specifying their parameters, and defining the interconnections between them. In an effort to allow non-programmers to configure their own system directly, a simple interface language is available. The language is declarative, meaning the statements may be rearranged in any order without altering the functioning of the system. Once written, the "script" file is interpreted to initialize the data structures for system execution.

6. State-Configured Layered Control

In Chapter II of this dissertation, it was stated that a primary drawback of the "pure" subsumption (layered) control approach advocated by Brooks [Ref. 31] lay in the inability to explicitly configure the desired mission. State-Configured Layered Control [Ref. 56] was developed in an attempt to address this shortcoming. Tests with an autonomous underwater survey vehicle [Ref. 151] indicated that the complexity of the layered control architecture increases significantly as the number of required behaviors increases. Furthermore, it was found that the overall vehicle performance is sensitive to subtle interactions between the concurrently-executing behaviors.

To overcome these unanticipated interactions, a higher level of control was added to guide the behaviors. This is accomplished by partitioning, or *configuring*, the layered control structure according to the current phase of the mission in an attempt to minimize the number of behaviors active at a given time. Determination of the current state is provided by a state transition table, which accounts for all possible states and the transitions between them. Each state determines which behaviors are active, establishes the parameters required by each, and specifies the priorities necessary for conflict resolution.

While State Configured Layered Control acknowledges the need for a higher level coordinator of layered behaviors, the state transition table presents its own problems with respect to modification [Ref. 152]. At present, a change in mission requires a complete reconfiguration of behaviors, involving the construction and validation of a new state transition table.

This architecture has led to the forward-chaining variant of RBM, henceforth labeled RBM-F. The state transition table can be employed as a conceptual tool for the development of the top level control software, implemented in a forward-chaining, rule-based language. RBM-F is examined in detail in Section D of this chapter.

7. NASREM

An elaboration of Saridis' three level model, NASREM actually encompasses a precisely defined six level software hierarchy for telerobotic applications [Ref. 43]. The NASREM architecture has since been generalized to permit application to a software system for the cooperative control of multiple AUVs [Ref. 153]. NASREM is an example of a purely hierarchical approach, in which behaviors contained at one level are initiated only when directed by the next higher level of the hierarchy. Together, these levels are said to form a *command hierarchy*. In addition, each level operates at a time scale longer than that of the level immediately below it in the hierarchy. The timing characteristics of each level thus form a *temporal hierarchy*. Each level consists of a set of closed-loop control systems containing sensory processing, world modeling, task decomposition, and performance evaluation unique to that level. Levels communicate with each other via global memory. The command and temporal hierarchies of NASREM are also to be found in RBM, although interlevel communication is much more restrictive in the latter.

Even though all interfaces are well defined, the NASREM architecture is somewhat inflexible, leading researchers to implement subsets and variants of the standard. This situation has caused the NASREM standard itself to evolve in an effort to become more acceptable to the research community. [Ref. 55]

Figure 7 pictorially shows the relationship between several of these software control architectures and RBM. An architecture connected to another on the left by a solid line is a subclass of the more general architecture. Much significant research involving control software designed around levels of conceptual abstraction has evolved from Saridis' original work [Ref. 37]. The subclass of architectures labeled SSS took Saridis' model a step further by specifying the time and space domains in which each level operates. The Rational Behavior Model refines SSS by associating a particular programming paradigm to each of the three levels while slightly refining the time and space domains at each level. The two subclasses of RBM, RBM-B and RBM-F, defined later in this chapter,

are distinguishable by the approach taken to the implementation of their respective top levels.

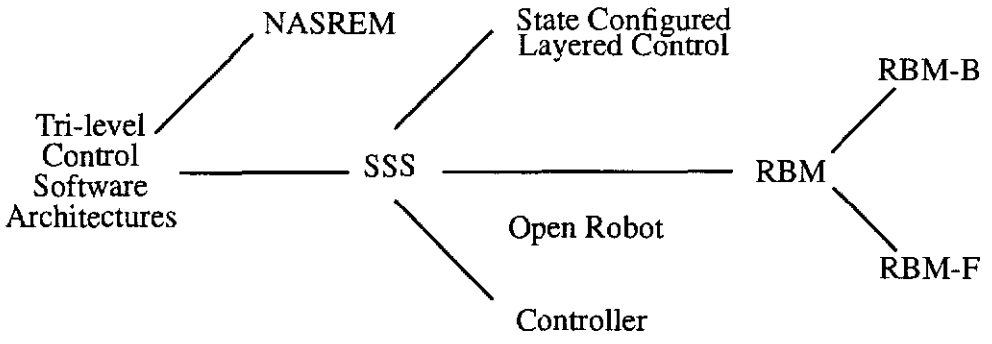


Figure 7. Tri-level Software Control Architectures

C. FUNDAMENTAL CONCEPTS

Before a formal discussion of any software system begins, a clear, concise definition should be given for all terms and concepts which might introduce confusion or ambiguity. Too often in the field of control software for autonomous vehicles this requirement has not been adequately addressed, resulting in a general lack of consistency of terminology among researchers. In this section, definitions for the concepts needed for the subsequent formal specification of RBM are given. Additional definitions will be introduced throughout the remainder of this dissertation as the need arises.

Complex systems are often organized into hierarchies to enhance efficiency and to focus the control of the problem-solving process. Given such an organization, complex problems can be solved most efficiently by successively decomposing the problem into increasingly simpler subproblems. More than one level of the hierarchy may join together to accomplish this, with the higher level forwarding its results to the lower level. At some point, the resulting components of the decomposition may be readily represented algorithmically. Such algorithms, if properly constructed, can produce clear, correct, and unambiguous in-

structions which, when carried out by the lowest levels of the hierarchy, will collectively result in the solution (accomplishment) of the initial problem (root goal).

DEFINITION: A *level* of control is characterized by a conceptual abstraction, of which temporal, spatial, and command hierarchies are the most common.

In a manner similar to that used by organizational hierarchies, a complex control problem may be best solved by decomposing it into smaller, simpler parts. Initially, the parts into which the problem is broken are themselves abstract problems; that is, little thought is given to the detailed functionality characterizing these abstractions. Instead, the concern is to successively decompose the problem into a finite set of simple subproblems which lend themselves to easy implementation. Each component subproblem created in this way identifies a goal to be attained on the way to solving the overall problem.

DEFINITION: As *Problem decomposition* of the root goal proceeds, the intermediate and primitive goals may be placed in a tree structure to assist in the orderly search for a problem solution.

DEFINITION: A *goal tree* is a graphical representation of AND/OR goal decomposition, with the root node representing the root goal, the leaf nodes representing the *primitive* goals, all other nodes representing intermediate goals subject to further decomposition, and the connecting arcs representing the logical relationship between subgoals and the goal from which they were decomposed.

After the problem has been decomposed into its constituent primitive goals, efforts can be directed to the satisfaction of these goals. These efforts, initiated by a primitive goal, typically involve the coordination of one or more behaviors.

DEFINITION: A *behavior* is an algorithm designed specifically to generate the numeric input required by a feedback control system which will, in turn, produce a change in the underlying physical plant consistent with the desired primitive goal which activated the behavior. The term *task* is a frequently used synonym [Ref. 18][Ref. 58].

The satisfaction of a primitive goal may entail the execution of a set of related behaviors. In some cases, behaviors may execute concurrently. In any case, the output generated

by these behaviors is numeric data in the form of *setpoints* and *modes*. These data collectively form the parameters required by servo loops to produce electrical signals that drive the associated servo controllers and actuators of the underlying physical system.

Another approach to autonomous vehicle control involves problem decomposition based on a set of task-achieving behaviors. Conceptual levels are discarded in favor of layers of control. Solutions are formulated through a subtle interaction of these layers.

DEFINITION: A *layer* of control represents the realization of a single behavior or competence of the underlying system.

A complete control software architecture may be built which comprises one or more of these layers. More complex levels of competence are achieved as additional layers of control are added. A layer, once activated, subsumes the role of the simpler layers lying at logically lower competence levels than the activated layer.

D. SPECIFICATION OF THE RBM SOFTWARE ARCHITECTURE

Like the examples of tri-level software control architectures described above, RBM utilizes the principle of abstraction to simplify the problem of mission control for an autonomous vehicle. In fact, RBM utilizes several complementary abstraction mechanisms in each of its levels. The three levels of RBM, from highest to lowest degree of abstraction, are called *Strategic*, *Tactical*, and *Execution* levels, respectively [Ref. 147]. Although these terms are not unique to this work [Ref. 18][Ref. 148][Ref. 154], they emphasize the conceptual abstraction of the levels when viewed as a whole. Defining characteristics of each level are listed in Table 1 for ready reference and are described in greater detail in the following sections.

The autonomous vehicles on which RBM is likely to be implemented will be required to accomplish missions that are best solved from a top down, goal-driven perspective. In recognition of this, the process of goal decomposition is handled naturally within the RBM hierarchy. The root goal, typically in the form of an overall mission objective, is decomposed within the Strategic level. The resultant primitive goals of the Strategic level then ac-

Table 1: CHARACTERISTICS OF RBM

Strategic Level
• Symbolic computation only; contains mission doctrine/specification • No storage of internal vehicle or external world state variables • Rule-based implementation, incorporating rule set, inference engine, and working memory (if required) • Non-interruptible, not event-driven • Directs the Tactical level through asynchronous message passing • Messages may be either commands or queries requiring YES/NO responses • Operates in discrete (Boolean) domain independently of time • Building block: the goal • Abstraction mechanisms: goal decomposition (RBM-B) and rule partitioning (RBM-F); both based on goal-driven reasoning
Tactical Level
• Provides asynchronous interface between Strategic and Execution levels • Behaviors (tasks) reside here and may execute concurrently • Behaviors are implemented as methods of objects • Primitive goals activate one or more behaviors • External interface of the model consists of two parts: the behavior activations from the Strategic level and the command/telemetry paths to/from the Execution level • World and Mission models maintained here • Responds to Strategic level with logical TRUE/FALSE • Setpoints, modes, active sensor commands, and non-routine data requests are output to the Execution level • Not interruptible except for data transfers; hard deadlines cannot be guaranteed • Operates in discrete event/continuous time domains • Building block: objects with behaviors • Abstraction mechanisms: class and composition hierarchies
Execution Level
• Numeric processing only • Responsible for software to hardware interface, underlying vehicle stability • All synchronous (hard real-time) processes reside at this level • Sensor data processed to specification of Tactical level • Servo loops run continuously and concurrently, synchronized by timed interrupts • Operates in continuous space/time domains • Building block: servo loops and signal processing algorithms • Abstraction mechanisms: loop composition, sampling frequency, and data smoothing.

tivate behaviors contained within the Tactical level. These behaviors are designed to produce the actions required by the primitive goals, in part by generating the commands necessary for the proper operation of the servo loops[1] located in the Execution level, and in part by manipulating the state of the tactical level and comparing it with predetermined mission subgoals and decision points. Finally, at the Execution level, the servo controllers are directed by the servo loops to manipulate actuators which cause changes in the relationship between the physical vehicle and its external environment. Various sensors then collect data to be ultimately used by the deliberative process contained in the Strategic level to guide future actions.

1. Strategic Level

It is apparent that the user of an autonomous vehicle must have an effective means of expressing a mission to the vehicle and, to some extent, the desired steps for carrying out the mission [Ref. 23][Ref. 55][Ref. 56]. The strategic level of the RBM architecture addresses this need directly by containing the explicit, high-level logic required to control an underlying robotic platform and the mission it is to perform. A customer who employs robots also requires that they behave predictably and efficiently; that is, the robot is expected to behave *rationally*. Seen computationally, global rational behavior is the realization of a deterministic sequence of simple actions resulting in some (possibly unobservable) side effects. Much recent research into so-called behavior-based autonomous control [Ref. 50][Ref. 52][Ref. 56] has achieved the desired global behavior by prioritizing primitive behaviors designed to compete with each other for the control of resources. A particular global behavior may be seen to emerge from these interactions, although typically at the expense of efficient operation and software modifiability. This latter weakness is a consequence of the global distribution of mission logic throughout the various behavioral layers comprising the architecture.

---

1. Servo loops typically include both hardware and software. The term Execution level is reserved in this work for the software component of such loops only.

The hierarchical approach to autonomous vehicle control emphasizes a more procedural approach to system behavior sequencing [Ref. 43][Ref. 155]. However, these systems also tend to suffer from the distribution of behavior-initiating logic throughout the architecture. Furthermore, the use of global blackboard-based data structures for interlevel communications is susceptible to undesired side effects linked to the execution history of the system [Ref. 99].

The primary strength of RBM, and one of its main contributions to the field of autonomous vehicle control, is the containment of behavior-sequencing logic at the upper, strategic level. The behaviors required for mission accomplishment are identified through the process of goal-driven decomposition. If the initiation of behaviors is expressed within a set of rules written in a rule-based programming language, the sequence of behavior activations can be controlled in a deterministic way through the use of a rule interpreter. In this approach, the function of insuring proper sequencing falls on the mission implementor.

Because of the diverse educational and technical backgrounds which will likely characterize mission designers employing RBM, it is deemed essential that the mission expressed at the Strategic level be easily read, understood, and modified in response to new mission requirements. Reordering of behavior activations, introduction of new vehicle capabilities, and need for a finer degree of top-level control are all issues directly affecting mission logic. In support of these requirements, and with an eye on involving the nonprogrammer in application development [Ref. 156], the implementation of the Strategic level of RBM must follow some specific guidelines and restrictions. These important characteristics are listed here, along with the rationale supporting them.

The basis for Strategic level design is the top-down decomposition of the mission based on goal-directed reasoning. As discussed in Chapter IV, this process involves the successive refinement of a root goal into constituent subgoals, continuing until simple, primitive goals are identified. These goals are not amenable to further simplification; instead, direct implementation is possible at this point. In this way, reasoning is restricted to the Strategic level with implementation relegated to the Tactical level. Because the

reasoning applied to the problem solution begins with the final goal and moves backwards to the circumstances supporting the solution, this process is called goal-directed reasoning [Ref. 115]. This approach to problem solving is best suited for autonomous vehicle control because all tasks and the sequence of their execution are explicitly identified. This is essential if the user of such a vehicle is to have confidence in the system and its abilities.

Since the reasoning process typically results in a series of logical steps leading to problem solution, it is natural that the specification of the Strategic level be expressed in a manner suited to represent concepts of logic. Rule-based programming languages are ideally suited to this need and thus are to be utilized at this level of RBM. An obvious advantage is that the language is executable, averting the need for a separate translation of the mission specification into another language. A translation of this type would also forfeit the conciseness of the rule-based language while introducing complexity.

During this phase of mission development, the use of state variables other than those necessary to support the reasoning process are to be avoided. This restriction has the primary purpose of insuring the simplicity and understandability of the mission logic and to enforce separation of the goals and their implementation. The Strategic level is responsible for determining the sequence of actions the vehicle should take, whereas the Tactical level is responsible for the implementation of those actions. As a consequence, computation at this level is purely symbolic. This is in contrast to the numerical processing which occurs at the lower levels of the architecture.

Ultimate control of the vehicle resides with the Strategic level. The search of the rule set results in a call to the Tactical level for an initiation of a behavior. Each call is an instance of one of two types: a *query* or a *command*. A query is a request for information; however, because of the restriction on the use of state variables at the Strategic level, replies to these queries must be in binary form (i.e., TRUE/FALSE). Commands, on the other hand, are directives which expect no response other than an acknowledgment that the command either has been accepted or carried out to completion. Any feedback which may be subsequently required for decision-making is obtained through a process of polling

using queries. Again, simplicity and the isolation of reasoning from implementation are factors supporting this restriction.

This emphasis on simplicity, at the expense of a greater degree of control, also requires that the Strategic level be free of semantic parallelism. Humans generally are better able to express concepts sequentially rather than in parallel. By placing this restriction on the Strategic level, the overall understandability of the implemented mission is assured, confidence in the rational global behavior of the system remains high, and subtle, undesirable effects caused by parallel interactions within the program are avoided. The benefits of parallel execution are not precluded, however, given that the parallelism is provided by the compiler/interpreter of the rule-based system.

The rule interpreter operates independently of time. Therefore, the sequence of rule firings is driven solely by the search of the inference engine. When a primitive goal is encountered, the Strategic level waits for acknowledgment from the Tactical level before proceeding. Also, because the flow of control at the Strategic level depends on the logical truth or falsity of queries, operation at this level is in the discrete Boolean space domain. With the addition of the search mechanism imposed by the inference engine, the Strategic level acts as a sequential circuit, with the memory representing the state of the search. Put in this perspective, the expression of the mission becomes an exercise in propositional calculus (albeit with lazy evaluation), and the complexity of unification is avoided.

A further characteristic of the Strategic level is that it be non-interruptible from within the RBM architecture. A thread of reasoning, realized through rule chaining, must be allowed to proceed to completion (i.e. to terminate at a primitive goal). Hence, the Tactical level cannot send commands to the Strategic level. Changes in mission state and vehicle environment are to be reported only when requested by the Strategic level.

Given the above constraints, it is incumbent upon the mission specialist to strive for an accounting of all non-time critical situations in which the vehicle may find itself. This is done using a common approach whereby categories of anomalies are identified which require similar responses. A one-to-many mapping is done between the members of

the finite set of responses and the anomalies contained in a category [Ref. 143][Ref. 144]. To ensure full coverage, a primitive goal initiating a default behavior (surface, stop in place, etc.) should be included as a "last resort" alternative for each mission phase.

Certain circumstances warrant immediate attention, however. Avoidance of collision or loss of the ability to maneuver are examples of conditions which must be responded to in a manner outside the normal query-decision-command cycle. These situations are best accommodated through the initiation of reflexive behaviors. This class of behaviors is designed to override existing control to avert problems affecting the safety of the vehicle. The Execution level, operating synchronously and responsible for the direct manipulation of the vehicle's transport mechanisms, is best suited to carry out these behaviors in a timely fashion. Once initiated, control of the vehicle is wrested from the Strategic level until normal conditions are reestablished. It is conceivable that a return to total normalcy with respect to mission accomplishment is impossible, as would occur, for example, if a control plane on an autonomous submarine became inoperative. The degree of fault tolerance and recoverability of the vehicle in these circumstances depends on the application and design philosophy of the mission specialist. In any case, the mission logic contained at the top level must account for unanticipated events, either explicitly, as would be appropriate for mission replanning, or for cases when orders are disobeyed as a result of a reflexive behavior.

Another scenario to be addressed relates to systems employing distributed multiprocessors as hosts for each level of control. Should communications between levels be lost, or if the commands or data from an adjacent level is determined to be erroneous, behaviors should be available to insure, at a minimum, the integrity of the vehicle. This is made possible through the isolation of responsibility within each level. The Execution level maintains the stability of the vehicle at all times. If commands from the Tactical level are interrupted, the vehicle remains stable. It may also be desirable for the Execution level to perform a recovery maneuver should commands not be received after a period of time. Likewise, the Tactical level should be endowed with behaviors whose purpose is to provide

sufficient direction to the vehicle following the loss of the Strategic level. In this way, hardware failures, while possibly jeopardizing the success of the mission, may not necessarily result in the loss of or damage to the vehicle.

The potential for human intervention may be required in certain circumstances. Thus, even though the Strategic level cannot be interrupted from within, the possibility exists for external interruption if the inference engine executes in an interpretive mode, as opposed to a compiled executable module. Just as there are rules that query the Tactical level about the status of various vehicle subsystems, a rule may be included in the rule set to provide a check for receipt of messages from external sources. The success of such a rule could result in the activation of additional stored rule sets or perhaps even the receipt of a new mission. Interruptions of this type would typically be related to recall, mid-mission reconfiguration, or fail-safe procedures.

The building block of the Strategic level is the goal. The logic of mission accomplishment through behavior sequencing depends solely on the relationship of these goals with respect one another. Individual goals are identified through goal decomposition based on goal-directed reasoning, concepts discussed in detail in Chapter IV. Once the set of primitive goals has been identified, the behaviors required to satisfy these goals will likewise be specified. The next step in the construction of the Strategic level is to express the logic in a symbolic, rule-based language incorporating a rule interpreter. The result of the interpreter's search will be the expected sequence of primitive goals and their associated behavior activations.

Originally, the strategic level of the RBM was defined to contain a concise, operational doctrine which described a top-level control strategy based on the avoidance of behavioral conflicts as opposed to the resolution of these conflicts [Ref. 147]. Specifically, the doctrine expressed the decomposition of the overall mission in terms of a goal tree with constraints on its traversal. Alternative branches of the goal tree could be combined to form an AND/OR search graph (see Chapter IV). This graph was then searched in depth-first fashion by a backward-chaining inference mechanism. The inherent ordering of the rules

as they were searched insured the avoidance of conflicts. Modifications to the overall behavior of the system could be accomplished by altering the logic of the rules or, more subtly, by changing the rule order [Ref. 23].

This approach to the executable representation of operational doctrine is a very powerful concept and as such will be retained in this work in its entirety. The vehicle for which the original RBM was developed, the six-legged Adaptive Suspension Vehicle (ASV) [Ref. 44], was not autonomous in the purest sense, however. The Strategic level of the RBM employed on the ASV was primarily responsible for the coordination of limb motion based on free (non-periodic) gaits and consistent with some desired motion goal. These goals, in the form of vehicle motion commands, were provided by a human operator by means of a joystick. This configuration allowed for human control of three major axes of motion: forward velocity, lateral velocity, and turning velocity [Ref. 44].

Thus, the human operator of the ASV generated the high-level goals to be attained by the vehicle and provided the sequencing of those goals in support of mission accomplishment. The ASV's Strategic level was then tasked to control leg motion in such a way as to satisfy the joystick command. The rule set describing this coordination was fixed, in that different missions did not require alteration of the Strategic level. Therefore, the term *doctrine* was applied to the code of the Strategic level, following the definition used by the military when referring to well-understood, well-documented strategies for the accomplishment of specific tasks.

In the domain of autonomous vehicles, no human is available to provide the functions of determining goal attainment, identifying unreachable goals, and selecting alternative goals. These capabilities must be part of the Strategic level. In addition, these issues are mission related; that is, each mission may have unique success and failure criteria for the constituent phases of the mission along with detailed contingencies for each. The rules that contain this knowledge cannot be considered doctrine in the same sense as the ASV's Strategic level doctrine. Instead, this portion of the Strategic level is collectively referred to as the *mission specification*. Whereas military organizations rely on doctrine to

realize generic functions, an *operations order* is issued to specify a unique mission tailored to some set of circumstances. Together, the operation order and the field manuals (doctrine) provide the guidance essential for the proper execution of a mission by a military unit [Ref. 157]. Likewise, the mission specification and the doctrine make up the entire rule set found in the Strategic level of a RBM software architecture and contain the knowledge necessary for the autonomous vehicle to carry out its mission.

DEFINITION: The *Mission Specification* is the part of the Strategic level containing the rule set embodying mission specific knowledge.

DEFINITION: The *Doctrine* is the part of the Strategic level containing the mission-independent rule set containing the logic required to solve problems not unique to the mission at hand. Doctrine is in general vehicle dependent.

The addition of a mission specification to the doctrine does not alter the fundamental relationship between the Strategic and Tactical levels. This is not to say, however, that the doctrine is immutable. If a new mission involves strategies not addressed in the existing doctrine, the doctrine should be updated to reflect this. The analogy of military doctrine is consistent with this possibility, as all published guidance produced by the military is subject to review and revision to reflect changing missions.

The abstraction that links the mission specification to the doctrine is goal decomposition. At some point, the subgoals derived from the top-level mission goal will merge with the top-level goals of the doctrine. It should be emphasized that the inference engine used to search the composite rule set will not distinguish between the doctrine and the mission specification, and all rules will be represented within the scope of a single AND/OR goal tree. That is, the separation of rules into mission specification and doctrine is strictly for the purpose of human convenience and understanding.

Because of the goal driven nature of this approach to the construction of the Strategic level, it follows that a backward chaining inference engine would be the ideal choice for an RBM implementation [Ref. 158]. Although this approach is indeed well suited for the large class of potential missions characterized by a backward reasoning

solution, the strategic level of RBM is not constrained to this one implementation paradigm. The forward chaining approach to mission implementation is commonly used in autonomous vehicle control applications [Ref. 45][Ref. 50][Ref. 56]. For this reason, we introduce two classes of RBM, distinguished by the type of chaining employed by their respective strategic level reasoners. The backward chaining, doctrine-based implementations of RBM are instantiations of the class RBM-Backward.

DEFINITION: The *Rational Behavior Model-Backward* (RBM-B) is a form of RBM characterized by a backward-chaining inference mechanism at the strategic level employing goal decomposition and an AND/OR goal tree as the basis for its search.

The Strategic level of a member of the class RBM-B contains a rule set, composed of a mission specification and a doctrine, which essentially reflects the compiled logic a human operator would use to solve problems related to mission accomplishment; and a backward-chaining rule inference engine responsible for the orderly and logically consistent interpretation of these rules.

The search performed by the inference engine is likely to involve repetition, since values returned from the Tactical level can change over time. In the context of mission execution, some search paths are designed to be continuously tried until success is reached. The notion of *dynamic* search is an extension of the usual notion of AND/OR graphs as static data structures [Ref. 137]. In applications involving autonomous vehicles in dynamic environments, dynamically changing data must be assumed.

A second class of the Rational Behavior Model, RBM-Forward (RBM-F), includes implementations of the Strategic level still founded upon goal-driven reasoning but controlled using forward chaining. Forward chaining systems require that all states of the mission and transitions between these states be explicitly identified [Ref. 159][Ref. 160]. This class of software control architectures, RBM-Forward, is now introduced.

DEFINITION: The *Rational Behavior Model-Forward* (RBM-F) is a form of RBM characterized by a forward-chaining inference mechanism at the Strategic level employing a state transition diagram to organize the sequence of allowable state transitions.

The strategic level of a member of the class RBM-F contains three elements: (1) a rule set containing the rules which define the preconditions and postconditions associated with each transition between the members in the set of mission states; (2) a working memory containing the mission and vehicle variables necessary to uniquely identify the current state of the mission; and (3) a rule interpreter (a forward-chaining inference engine) which searches through the rule set, activating rules as the mission state warrants, and selects and fires a rule based on a well-defined conflict resolution strategy.

The rule set found in the knowledge base of an RBM-F architecture does not share the explicit hierarchical structure of the AND/OR graph found in the RBM-B mission specification and doctrine. Hence, RBM-F mission implementations are, generally speaking, less concise and less understandable than the missions expressed in the RBM-B Strategic levels. However, the forward-chaining inference mechanism associated with the RBM-F class does employ a search structure in its execution of the mission. This structure is a state-transition diagram (STD), augmented with transition path priorities. An explanation of the STD is given in Chapter IV of this dissertation.

The most general mission of the RBM-F controller is described by a traditional STD, which we call the mission STD. Here, the mission designer can configure the desired mission into phases, each phase represented by a state. Hence, the mission STD represents a top-level model of the mission and identifies the various phases of the mission (i.e., transit, search, track, etc.). These phases are subjected to a refinement process, whereby each state is "exploded" to reveal another STD. The lower-level STD contains the states internal to the parent state, including the appropriate conditions and actions for each transition. In many respects, the mission STD and the immediate partitions derived from it correspond to a mission specification of RBM-B. Beyond this point, the states identified through further partitioning correspond to the mission-independent doctrine of RBM-B. Eventually, the recursive partitioning of states results in the identification of final states. A final state represents the conditions necessary to perform interaction with the Tactical level.

It should be stressed that a final state in RBM-F is not equivalent to a primitive goal in RBM-B. A final state represents the set of conditions which must exist prior to the activation of behaviors in the Tactical level. These activations occur in concert with the firing of a transition out of the final state. In contrast, the primitive goal of RBM-B explicitly invokes the Tactical-level behavior from within the node. For this reason, additional states are needed for an exact translation between the AND/OR and STD structures.

Conventional State Transition Diagrams require that state transitions be identified for all possible conditions and that these conditions be exhaustive and mutually exclusive so as to avoid nondeterminism [Ref. 136]. A state transition diagram describing a mission based on the forward-chaining RBM-F paradigm may involve states which have multiple successor states but transition conditions which are not mutually exclusive. This gives rise to a conflict, the resolution of which must be accomplished to insure the proper behavior is initiated. Since traditional STD notation does not make allowance for this, a prioritized state transition diagram is required to resolve these conflicts. Usually, conflicts that arise between competing behaviors are handled through a system of voting or the prioritizing of transition paths with respect to each other [Ref. 31][Ref. 52].

The state transition diagram with path priorities provides a convenient, graphical tool for modelling systems which exhibit "competing" entities. The term *competing* is used here to describe a scenario whereby two or more processes/tasks/behaviors request some resource simultaneously. Competing entities could then be represented by several paths emanating from the same state and with non-mutually exclusive transition conditions. Previous attempts have been made to partition conflicting behaviors through the use of state transition diagrams, most notably Bellingham's State Configured Layered Control [Ref. 56]. This model utilizes a single, top-level STD to partition the mission in much the same way as the mission STD is used. Further partitioning of the State Configured Layered Control STD is not performed, however. Instead, each state of the STD defines a subset of a layered control architecture [Ref. 31]. Hence, State Configured Layered Control does not

address conflict resolution at the same logically abstract level used to define the mission. Therefore, this approach suffers from the verifiability problems characteristic of behaviorist systems [Ref. 55].

Because the implementation of the reasoning used in the design of the Strategic level in both RBM-B and RBM-F is guided by some form of search graph, the inference engine must keep track of the current phase of the executing mission. The inference engine can be thought of as manipulating the state of the mission. This state refers to the facts available for use by the inference mechanism in its search of the rule set. Since backward chaining systems use a search algorithm based on rule position and depth-first traversal, no explicit state is required to determine rule priority. Forward chainers, conversely, apply search techniques which do not necessarily rely on positional relationships between rules. Instead, an explicit database of conditions, or facts, must be maintained. This fact base, representing the current state of the mission, determines which rules are to be activated at a given time. In any case, it is essential that the mission specialist be familiar with the particular search strategy employed by the inference engine. Mission development can then proceed based on this strategy and with the understanding that the inference mechanism will maintain the state of the search.

The purpose of avoiding an explicit fact list (and global memory, for that matter) in RBM-B implementations and minimizing such a state in RBM-F implementations is to simplify mission development by abstracting the control problem to the maximum extent possible. Outside of the capability of expressing a mission in some rule-based language, it should not be presumed that the human mission specialist is a computer scientist. In addition, comprehensive knowledge of the underlying computer systems architecture by this individual should not be assumed.

The concept of the Strategic level of RBM is built upon the observed need for high-level logical control of complex autonomous systems. While this requirement has been met, the equally important issues of expressive power, understandability, and modifiability of mission logic have been addressed. Placing the constraints discussed

herein on the development of the Strategic level, while dramatic, is perceived as necessary if order is to be brought to the chaos of the "real" world. Once high level goals have been identified and their sequence of attainment specified, the vehicle must have the capacity to respond. This level of control is contained within the second of RBM's three levels, the Tactical level.

2. Tactical Level

The middle level of RBM, like the middle levels of the three-level architectures mentioned earlier in this chapter, acts as intermediary between a knowledge-based behavior sequencing mechanism and the lowest level vehicle-control subsystem. A wide variety of approaches have been applied to the design of this level, from the competing behaviors of subsumption in SSS [Ref. 149] to discrete event modelling using Petri nets [Ref. 148]. The Tactical level of RBM further refines concepts from both these philosophies. Specific characteristics of this level are presented here, along with the justification supporting each. As with the Strategic level, certain restrictions are included as part of the architectural definition.

The primary purpose of the Tactical level is to provide asynchronous coordination between the symbolic-based, behavior-enabling goals of the Strategic level and the numeric-based servo loops of the Execution level. To accomplish this, the Tactical level implements a finite set of *behaviors*. The result of a behavior may be a change to the internal state of the Tactical level, receipt and analysis of sensory data passed from the Execution level, a non-routine data request, or the transmission of commands in the form of numerical setpoints and modes as required for the proper operation of the Execution level subsystems. In addition, upon completion of a behavior, the Tactical level must respond to the Strategic level. This response may be explicit, as an answer to a query, or simply a return of control following the acceptance or completion of a commanded behavior.

The behaviors contained within this level of the model are non-logic based, repetitively executed processes, as compared to the rule-based reasoning of the top level. Behaviors may be thought of as being performed by one or more entities within the Tactical level. These entities, which possess a unique, defining, and persistent state, are referred to as *software objects* [Ref. 88]. A software object is essentially a finite-state machine producing a consistent output when an identical sequence of inputs is encountered, assuming the same starting state for each trial. A software design composed of a set of objects communicating with each other through well-defined interfaces is called object-based or object-oriented[2] [Ref. 89]. Object-based and object-oriented design methodologies are explained in [Ref. 88] and [Ref. 94]. Formal specification of each object and the behaviors associated with each may eventually be accomplished using an approach similar to the Spec *machine* construct [Ref. 161].

Because the behaviors required of an autonomous vehicle resemble the functioning of a human crew on manned vehicles, an object-based design methodology whereby the functionality of objects correspond to that of individual crew members may be adopted for the Tactical level of RBM. This approach was used in the instance of RBM presented in the next chapter of this dissertation. Implementation of the various behaviors defined for the mission are divided among these objects. Some behaviors may require interaction between two or more objects. This interaction, as specified by the object-based design paradigm, takes the form of message passing. Receipt of a message results in the execution of a corresponding method defined within the receiving object.

To enhance modularity and maintainability of the software, the objects of the Tactical level are organized in an *object hierarchy* [Ref. 88]. Child (dependent) objects are components of their parent object and can be accessed only through methods defined by the

2. Object-oriented programming involves the concept of class inheritance; object-based programming does not. Either is suitable for the implementation of the objects of the Tactical level. Both provide the advantages of ease of object instantiation, internal data and process encapsulation, and message passing. Object-oriented programming provides the additional feature of the class hierarchy, allowing the reuse of both variable and function definitions through the mechanism of inheritance.

parent. A child object has knowledge of its parent; hence, messages requesting services not contained within the child or its descendents must be sent to the parent for subsequent dispatch to other branches of the hierarchy. An exception is that data of a global nature (i.e., data required by several objects in the performance of their functions) may be obtained directly from an object designated to manage that data.

The object residing at the top of the Tactical level object hierarchy has no parent and must serve as the interface between the hierarchy and the non-object-based world. All behaviors implemented in the Tactical level originate at this object. Behaviors are responses to primitive goals specified by the Strategic level. Therefore, the Strategic level communicates to the Tactical level solely through the interface provided by this object. By way of analogy, the captain of a ship directs which goal is to be achieved. The goal is passed to the ship's Officer of the Deck (OOD), who has the responsibility to insure that the goal is attained. The OOD assigns tasks to subordinates in support of the main goal. When completed, the OOD reports to the captain and awaits further directives. Note that the captain should not be involved in the implementation of behaviors, nor should he be involved in the direct control of the ship. In this scenario, the captain resides outside the realm of the ship's operation.

The Tactical level object hierarchy and the behaviors associated with it, together with the telemetry systems and servo control loops of the Execution level, constitute a functionally equivalent robot vehicle as viewed by the Strategic level. Behavior relating to navigation, system checking, sonar interpretation, and obstacle replanning are appropriate for the Tactical level. In order to isolate the reasoning process of the Strategic level from the vehicle-dependent Execution level, the Tactical level must provide for the complete monitoring and control of the Execution level while being guided by the primitive goals of the Strategic level.

One characteristic of the Strategic level is its lack of knowledge of the operational environment, both external and internal to the vehicle. The purpose of this is to enhance top level simplicity, re-enterability, and consistency. The Tactical level, however, must base its

are behaviors whose execution times are functions of the environmental complexity and quantity of data to be assimilated. Some objects may require synchronous operation, such as the sensory receiver. For the most part, however, asynchronous operation is the norm; therefore, any process or task directly affecting the stability or safety of the vehicle must reside at the synchronous Execution level.

The Tactical level implements the goals directed by the Strategic level by producing setpoints, mode changes, and data requests recognizable by control loops within the Execution level and which, when applied to these loops, will result in the desired change to the physical state of the vehicle. This transformation is accomplished through the initiation of behaviors manifested in the activation of objects. The objects contained in the Tactical level are organized into a tree structured hierarchy. By restricting inter-object communications to parent-child links, this hierarchy is inherently loop-free. Taken together, these characteristics enhance the understanding, maintenance, and software reuse at the Tactical level.

3. Execution Level

This level, called the "Servo" level within SSS [Ref. 149], "hardware control" by Saridis [Ref. 37], and "real-time" level by Blidberg [Ref. 39], is the best understood of the three levels of RBM. This derives from the fact that the body of knowledge constituting closed-loop servo control theory [Ref. 164] is mature, having developed over many decades. Hence, in most instances, the design and implementation of software at this level is accomplished by control engineers, not computer scientists. Indeed, this level is often taken more or less for granted by the researchers concentrating on the upper, more abstract levels of control.

The Strategic level of RBM replaces the supervisory control provided by the human operator in remotely operated and manned vehicles. In some instances, human involvement in the control of a vehicle goes so far as to include direct control of thrusters [Ref. 165]. This degree of control requires that the human ROV pilot possess highly

developed hand-to-eye coordination and motor skills, similar to those possessed by a helicopter pilot. For autonomous control of vehicles, these functions must be provided within the RBM architecture. It is appropriate for the Execution level to provide these capabilities.

Therefore, the Execution level of RBM must provide the means necessary to assure the underlying stability of the vehicle. Within the realm of autonomous underwater vehicles, this requirement is most easily satisfied through the application of at least the following: (1) a steering autopilot[3] for heading control or rate control; (2) a diving autopilot capable of providing stable depth changes or control over pitch angle on command; (3) a speed control autopilot to adjust the vehicle speed in either a vehicle speed control mode or propeller rate mode; and (4) a hovering mode autopilot for maintaining position in a prescribed attitude [Ref. 166]. Land and air vehicles require specialized autopilots to provide the necessary stability, but whose purpose are the same. The internal implementations of these autopilots is unimportant as long as adequate robustness and stability of vehicle motion is guaranteed.

By their nature, autopilots provide stability as long as the underlying servo loops are properly designed and continue to meet a prescribed update rate. This constraint may be met through the use of a fixed schedule triggered by timed interrupts from a real-time clock. Processes enabled in such a way are said to be time driven or *synchronous*. Rate monotonic scheduling guarantees efficient use of processor capacity in such circumstances [Ref. 167].

Commands must always be available for consumption by the autopilots. In vehicles operating in water or on smooth land terrain, these parameters may be provided by asynchronously executing objects in the tactical level. The robust nature of well designed

---

3. Autopilots are devices that typically consist of a servo loop driving a servo controller. Setpoints are the input parameters to these loops. The setpoints differ depending upon the control mode selected. So, for example, a steering autopilot may operate in a heading mode, requiring input based on a geometric angle offset or in a rate mode, requiring input related to the desired turn rate of the vehicle. Autopilots for depth and speed may likewise operate in different modes.

following an upgrade to a sensor or a change in the fundamental capability of the robot to perform a given task.

There still remain many important problems to be solved at the Execution level, especially for vehicles which operate in three dimensions [Ref. 174][Ref. 175][Ref. 176]. In contrast to the operation of wheeled vehicles in a laboratory setting, where the servo problem reduces to control of wheeled rotation [Ref. 54][Ref. 149], the seemingly simple task of remaining stationary with respect to a given coordinate frame (i.e., station-keeping) is a very difficult problem for underwater vehicles. Unmanned airborne vehicles have their own unique stability problems [Ref. 177].

E. SUMMARY

The Rational Behavior Model provides important advantages over existing approaches to software development for the control of complex, autonomous vehicles, while alleviating or ameliorating some of their distinct disadvantages. As such, RBM provides a more powerful and versatile framework for the construction of these increasingly complex software systems. Versatility results from the isolation of responsibility inherent in each level. This conceptualizing is carried over to the design of the architecture, primarily through the association of specific programming paradigms to each level. Selection of computing hardware, operating systems, and implementation language is guided by product availability and the experience of the designers.

RBM is also powerful with respect to the software maintenance of an existing system. Reasoning related to mission accomplishment resides only in the Strategic level; behaviors which carry out the goals of the Strategic level are located in the Tactical level; and synchronous processes responsible for stabilization of vehicular motion, hardware manipulation, reflexive behavior, and basic telemetry processing fall within the scope of the Execution level.

In addition, use of the object-oriented design methodology at the Tactical level enforces modularity, data and procedural encapsulation, and formalized interfaces within soft-

ware objects. Finally, modification of the software of one level need not affect adjacent levels. A well-defined, inter-level communications interface, terminated by single data "gateways" insures that commands and data are available to those processes requiring them.

VI. ARCHITECTURE VALIDATION

A. INTRODUCTION

This chapter discusses the practical issues related to the design and implementation of the Rational Behavior Model. Chapter V provided a description of the attributes which characterize an instantiation of RBM, independent of the application, vehicle, or computational configuration. By insuring that these attributes are not violated during implementation, designers of autonomous vehicles who choose RBM as their control software architecture may expect to benefit from the model's advantages while still being afforded a reasonable amount of flexibility in terms of developmental decision making. Of course, this flexibility will certainly not prevent the consequences of a poor design and the resulting unanticipated or unacceptable performance.

What follows is an example implementation involving the simulation of a small Autonomous Underwater Vehicle executing a real-world mission under RBM control. The discussion begins with a review of the issues a control software systems designer must confront when considering the resources required to best support the vehicle's capability and the class of missions that will be expected of it. These issues include the degree to which concurrency and real-time execution are needed; the available hardware resources and their configuration; and the programming languages and operating systems selected for the implementation.

As an introduction to the experiments and analysis which forms the rest of the chapter, the example mission is presented, followed by an explanation of the RBM instance constructed to realize this mission. The model is then tasked to execute the mission on a laboratory simulator. The various data resulting from this experiment are examined and evaluated, and conclusions are drawn.

B. IMPLEMENTATION CONSIDERATIONS

Development of the control software architecture for an autonomous vehicle is strongly influenced by the hardware and software resources on hand, the experience of the programmers, and the characteristics of the missions assigned to the vehicle. Resources in this context include the number and type of processors, memory, communications facilities, and other assorted hardware support; the capabilities of the operating system(s); the existence of pre-written software modules; and the availability of languages and software development tools. This section reviews the practical issues of implementing a software architecture with respect to languages, operating systems, real-time constraints, concurrency, and distributed computing.

1. Languages

The major programming paradigms were examined in Chapter III. While each paradigm has applications for which it is well suited, programmers of autonomous vehicles should not be expected to be "fluent" in every one. Additionally, the project's budget may not allow for the purchase of representatives of each class of programming language along with the associated compilers, editors, design tools, etc.

Nevertheless, the choice of RBM as a control framework implies a multi-paradigm approach. Besides the advantages of this architecture (as enumerated in Chapter V), this requirement is necessary to insure the desired isolation and separation of responsibilities between the disparate levels. The expressive power of each language and the potential for simplified software modification resulting from this design are sufficient reward to justify any initial "start-up" costs associated with the purchase and learning of a new language.

One additional potential cost involved with the use of multiple programming paradigms is designing the interface between them. The capability of invoking "foreign" language calls from within a program is not always provided for by the particular compiler/ interpreter on hand. Since each language has its own parameter passing and variable typing conventions, a separate interface must be built for each. For example, the version of Quin-

tus Prolog used for the implementation of the Strategic level described in this chapter provides pre-defined protocols to call C, Pascal, and Fortran routines [Ref. 178]. Since Ada was chosen as the language for the Tactical level, the Prolog-Ada interface had to be constructed from scratch. Furthermore, the Execution level was written in C; therefore, an Ada-C interface also had to be built. This topic will be elaborated in the discussion section of this chapter as it is an essential, although implicit, component of the architecture.

2. Operating Systems

The services provided by an operating system may be quite diverse and, as such, are often tailored to a particular application [Ref. 106]. Support provided by the resulting "executive" or "kernel" may include real-time job scheduling, multitasking, communications, memory management, and exception handling. Of course, operating systems are software programs themselves and therefore must share the memory and processor with the application programs. In the past, these resources were scarce, and the responsibility of providing these services fell upon the programmer. In addition, the control software was typically generated and compiled off-line and the resulting object files subsequently ported to the target computer. The programmer would then initiate the control program by loading the appropriate registers prior to execution [Ref. 179].

The need for efficiency, particularly in applications involving real-time execution requirements, has also resulted in software control system designs lacking a traditional operating system. Recently, however, operating systems have been introduced that explicitly address real-time concerns [Ref. 170][Ref. 180]. These products, combined with the simultaneous reduction in cost and increase in speed of hardware, have relegated the decision regarding operating systems with respect to autonomous vehicles to one of convenience rather than of performance.

Still, the actual operating system(s) selected will depend heavily on the requirements placed upon them. The DOS family of operating systems are general in scope and perform efficiently but do not provide multitasking. UNIX does support multitasking but

cannot guarantee the scheduling necessary to meet real-time deadlines. While less problematic, a version of the desired implementation language that is also compatible with an existing operating system may not be available. For the implementation described herein, versions of Prolog and Ada were available for use with UNIX but not for IRIX, the operating system of the Silicon Graphics Iris workstations. Hence, at a minimum, two separate processors and two different operating systems were needed for this instantiation of RBM. Of course, RBM is not constrained to any specific language/operating system combination.

3. Real-time Constraints

Processes whose measure of "correctness" is a function of time as well as acceptable output are classified, with the entities that they control, as *hard real-time* or, more simply, *real-time* systems. Should these systems fail to meet a deadline imposed upon them, even if the output produced is computationally correct, serious consequences involving vehicle integrity or damage to objects in the vehicle's immediate vicinity may ensue. Typically, real-time processes are used to insure the fundamental stability, prevent imminent collision, and respond to internal failures of the vehicle.

Within the RBM framework, all real-time processes reside in the Execution level. Once the various tasks have been identified and implemented, a scheduling policy must be generated. It is the responsibility of the real-time scheduler to insure that the processes are dispatched in a sequence which meets all deadlines.

The deadlines under which the real-time control processes of the NPS AUV execute are based upon the receipt of timed interrupts. Each interrupt signals a process to commence. It is therefore necessary for the previous process to have completed its tasks prior to the commencement of the next. By analyzing the execution times of each process separately along with the system "overhead", if applicable, the adequacy of the schedule to meet all constraints can be verified.

4. Concurrency

The definition of the Tactical level of RBM states that this level shall be composed of software objects that communicate via a message passing mechanism. A significant provision of this design is that some (or all) of the objects may be active at a given time; that is, several objects may embody separate, distinct threads of control. On a single processor, logical concurrency of these objects is realized through the "interleaving" of each task's execution under the guidance of a time-sharing or priority-based algorithm. If multiple processors are available to support true parallel execution, objects may run simultaneously. In either case, the actions of each are coordinated through the sending of messages to one another.

Concurrency may be provided in several ways. A multitasking operating system can emulate parallelism on a single processor. On transputers or other multi-processor platforms, the operating system or programmer can assume the responsibility for the efficient assignment of tasks to available processors, a technique called *load-balancing* [Ref. 107]. In this scenario, the benefits of concurrent execution must be weighed against the time spent dividing the problem and assigning each piece to a processor. Needless to say, load balancing is not an attractive alternative for ill-defined, unstructured problems.

Another approach to concurrent execution is to utilize control constructs provided by a concurrent programming language. This allows for the explicit identification of potential parallelism within the program, although multiple processors, and a scheduler capable of assigning the tasks to the various processors, will still be necessary to realize an increase in performance. Ada provides these features and was therefore selected for this work. Although not used in this particular implementation of RBM, it is expected that future extensions to this instantiation will call for concurrent processing.

5. Distributed Computing

One approach to the employment of multiple processors is through a distributed network of computers. As mentioned earlier in this section, the implementation of RBM

discussed in this dissertation required that a multiprocessor solution be used. The processors, in the form of separate workstations, were *distributed* in that they executed simultaneously with respect to each other. In this case, message passing occurred over communications channels provided by an Ethernet link [Ref. 181]. The processors in this network were *loosely coupled*, in that each had access to its own private memory (in contrast, a configuration in which separate processors share a common memory area is called *tightly coupled*). Inter-processor communication is a significant issue regardless of the configuration, but one for which solutions are generally available [Ref. 182].

A distributed system complements the emphasis placed by RBM on the isolation and divided responsibility of each level of control. As such, designs calling for the assignment of each level to a separate operating environment are feasible. For instance, a Strategic level written in Prolog, executing interpretively and running on a UNIX-based processing platform can pass commands to and receive replies from a Tactical level designed as an object hierarchy, implemented in Ada, and running on a second UNIX-based computer. Finally, this Tactical level can issue commands to an Execution level written in yet another language and hosted on a third processor. Alternatively, the Strategic and Tactical levels can both execute concurrently on a single processor while the Execution level resides on a separate host. The flexibility of this design is obvious, despite the implementation restrictions placed on it by the definition of RBM. The three processor network described above was used for the instantiation of RBM discussed in the following sections; the two processor configuration, on the other hand, reflects the design of the computing facilities on the actual NPS AUV [Ref. 72].

C. THE MISSION

During the spring of 1992, a workshop [Ref. 183] was convened at the Florida Atlantic University under the sponsorship of the National Science Foundation to discuss and advance the state of autonomy within the field of underwater vehicle technology. The workshop participants recognized the importance of cooperation and collaboration among the

members of the Ocean Technology community as a way to focus efforts toward the attainment of common goals. Among the recommendations presented in the workshop results was the use of inter-institutional technology demonstrations to evaluate the effectiveness of current research concepts. To this end, three sample AUV mission scenarios were selected, each containing significant challenges and sufficient realism to insure that even partial task completion would insure valuable experimental results. The three task scenarios selected were search and rescue, pollution source location, and navigation with obstacle avoidance. Each mission provides a realistic basis for the employment of autonomous underwater vehicles.

The scale and scope of these missions make them ideal for the experimental validation of any control software architecture. Each mission requires the demonstration of important capabilities expected of an AUV. Taken together, these capabilities include search, surveying, sampling, obstacle classification and avoidance, and payload management. Implicit in each mission is the obvious need to maintain vehicle stability, to navigate satisfactorily in a dynamic environment, and to provide provisions for the safety of the vehicle. For the purpose of this research, one mission, search and rescue, was selected as a test case for evaluation in this chapter. All of the missions are listed in Table 2

D. INSTANTIATION OF THE MODEL

This section discusses the implementation of RBM using a simulation of the NPS AUV as the underlying vehicle and the search and rescue scenario as the mission[1] to be accomplished. Strategic and Tactical levels were constructed for this research with this mission in mind. A non-linear, hydrodynamically accurate model of the NPS AUV has been previously developed jointly by students from the Computer Science and Mechanical Engineering departments of NPS in support of vehicle testing in the NPS pool [Ref. 184]. This simulation contains all the essential characteristics of an RBM Execution level, including

1. This mission has come to be known as the *Florida mission* because the demonstrations are scheduled to take place off the Florida coast.

Table 2: THE THREE FLORIDA MISSIONS AND THEIR REQUIREMENTS

Mission 1: Search and Rescue. Given the parameters of a search region, the AUV will traverse to the region, locate a subsurface buoy, cut the buoy's mooring line, drop a weight as close to the buoy as possible, return to the launch site, and surface.
Mission 2: Navigation and Obstacle Avoidance. The AUV transits to its starting position. At start time T, it transits to waypoint #1 where it releases a marker at T+5 min. The AUV proceeds toward waypoint #2 choosing to either: avoid the target area entirely; or to identify and locate the relative position of all targets within the target area, proceed to the position represented by the centroid of the shape formed by the targets, and drop a marker at that point. In either case, the AUV will arrive and drop a marker at waypoint #2 at T+15. The AUV will proceed towards waypoint #3, choosing to either: avoid the obstacle area altogether; or to enter the area and avoid the barrier. In either case, the AUV will arrive and drop a marker at waypoint #3 at T+20. The AUV will then return to its starting point, arriving as close to T+25 as possible.
Mission 3: Pollution Source Localization. The AUV will be launched from shore, downstream of the pollution source. The AUV will transit upstream using depth contour. The source of pollution (either acoustic signature or florescent dye) will be located and a beacon deployed. The AUV will then move to an exit point on shore.

models of the guidance, depth, and speed control autopilots, vertical and directional gyroscopes, and a synchronously updated control loop. Each level of the resulting RBM is described in detail below, and discussion concerning implementation choices and the rationale behind each is included when appropriate. To provide closure with the discussion on forward and backward chaining presented in Chapter IV, both forward and backward chaining Strategic levels for this mission have been implemented and are compared.

1. Strategic Level

The set of primitive goals associated with the Florida mission and the sequence of their execution results from an application of goal-driven reasoning by the mission developers. Each mission originates with a single high-level goal, such as search and rescue. This top, or root, goal is then reduced by introducing simpler but more specific subgoals. Taken together, these subgoals will satisfy the root goal. In the search and rescue mission, the

problem reduces to having the AUV traverse a path from the launch site to a specified location, search for and locate a subsurface buoy, perform a simple task to mark the location of the buoy, and return to the launch site. These subgoals are subject to further decomposition, depending on the degree of control desired at the top level. A Strategic level derived from an overly thorough refinement of the goals is analogous to a human micro-manager, whereas one based on a coarser decomposition and providing more general primitive goals to the Tactical level would reflect a "hands-off" management style. Of course, this would necessitate a more complex implementation of the corresponding behaviors at the Tactical level. In any case, the reduction proceeds until subgoals of sufficient simplicity result which lend themselves to implementation. This completes the reasoning portion of the mission planning process.

The next step is to implement this mission. Two possible approaches are investigated in this work: backward chaining and forward chaining. Since the Rational Behavior Model as defined can accommodate both approaches, an example of each is investigated.

a. The Backward Chaining Implementation

The goal-driven solution is naturally suited to a backward-chaining implementation [Ref. 46]. For this purpose, the logic programming language Prolog was selected. Prolog offers an additional advantage in that it provides a control mechanism called an *inference engine* to search the set of rules. As a result, the inference engine determines the sequence of primitive goals and, thus, the order of behavior activations. Although the Prolog programmer is not concerned with the details of the inference engine, how it works greatly affects the resulting structure of the program. This stems from the fact that Prolog uses depth-first search in its traversal of the rule set.

Prolog provides the basic constructs of facts, rules, and queries found in rule-based languages. These clauses can form the basis of sophisticated expert systems whose purpose is to reason about the problem and select the best solution from a set of alternatives. The logical relationships upon which these decisions are made are embodied in the Prolog

rules. Each rule expresses an *implication* or *if-then* relation. As shown in Figure 8, the *if* part of the Prolog rule, or body, lies to the right of the symbol ":-" and can consist of one or more subgoals linked through the logical AND operator. In Prolog, AND is represented by a comma. The *then* part of the relation, or head, lies on the left of the ":-" delimiter (read as "if"). Left sides of rules are restricted to one term only, a characteristic of the Horn clause form, as discussed in Chapter IV.

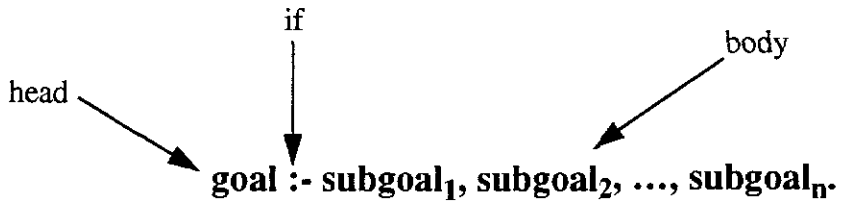


Figure 8. A Prolog Rule

The goal residing on the left side is considered to be satisfied only when all subgoals on the right side have been satisfied. The Prolog inference engine attempts to satisfy each of these subgoals in order from left to right. Because multiple subgoals are logically connected by AND, the failure of any one results in the failure of the rule. Evaluation of that rule stops when a failure is encountered, a technique known as *lazy* evaluation.

Rules which share identical left sides are related to one another through the logical OR operator. In this case, if one rule fails, the next rule will be evaluated, and so on, until a related rule succeeds or until no more rules sharing the goal remain. Again, lazy evaluation stops any further search of the related rules once the goal has been satisfied. Because Prolog scans the rule base from top to bottom, rules related by an OR should be ordered with respect to one another to reflect their respective priorities by placing the highest priority rule at the top of the group and the lowest priority rule at the bottom.

Prolog always attempts to satisfy a subgoal by matching it to a fact or rule head. When a match is found, the process proceeds to the next subgoal in the rule and an-

other match is attempted. The inference engine guiding the search marks each goal to provide a reference should the current inference chain fail. If a match cannot be made given the existing circumstances, an attempt to resatisfy the previous subgoal is automatically done through a control mechanism called *backtracking*. If no subgoal can be satisfied, the corresponding rule is skipped and an alternative rule is selected, if available.

Prolog provides the built in control flow primitive *repeat* which, when used in concert with backtracking, allows for the creation of loops. When first encountered, the repeat succeeds and the loop is entered. The repeat subsequently succeeds when encountered through backtracking. This provides for multiple attempts to satisfy those subgoals lying to the right of the repeat.

Another control primitive is required to insure a strict, iterative loop. This is the *cut*, denoted by "!", which acts to block backtracking. In the context of RBM, the cut is used to eliminate internal cycles and to force a specific sequence of behavior activations. While the use of these control flow predicates compromises the program from the standpoint of pure logic programming, they are essential if unwanted backtracking is to be avoided.

Reference is now made to the Prolog implementation of the Strategic level for the search and rescue mission given in Appendix A-1. Since a backward-chaining language is being used, this instantiation is a member of the class of RBM-B architectures. The program is initiated when the query "?-execute_auv_mission." is issued to the Prolog inference engine. Scanning the heads of each rule from the top of the rule set to the bottom, the rule "execute_auv_mission :- initialize, repeat, mission." is encountered. Prolog will first attempt to satisfy the subgoal "initialize". The rule set is scanned, again from the top, for a matching rule head. Two initialize rules exist, although the rule "initialize :- vehicle_ready_for_launch_p(ANS1), ANS1 == 1, select_first_waypoint(ANS2)." has priority because it is encountered first.

The subgoal "vehicle_ready_for_launch_p(ANS1)" is an example of a primitive goal; that is, it is a goal not subjected to further decomposition. Primitive goals mark

the interface between the Strategic and Tactical levels and may represent requests for information or commands. A request expects a response that can take on values representing TRUE or FALSE. This value in turn influences which one of several alternative reasoning paths is taken by the inference engine. A command, on the other hand, is a directive that may or may not expect a response other than an acknowledgment that the task was initiated or accomplished. The variable ANS (for ANSwer) associated with each primitive goal accepts the Boolean value returned from the Tactical level and is then available within that rule for comparison.

An implementation detail associated with the use of Prolog with foreign language calls is the requirement to supply return variables in every case. In the program under discussion, the primitive goals are actually making calls to C routines written to support the communications link between the two workstations hosting the Strategic and Tactical levels. The C routines then pass the desired goal over the network where it is accept by the Tactical level. Although some primitive goals are not followed by a comparison and that value will be discarded, the variable is still required as part of the function call acknowledgment.

After the primitive goal invokes the appropriate behavior, the Strategic level must wait until a response is received. The Strategic level is, in effect, suspended during this time. If the behavior associated with readying the vehicle for launch is completed successfully, a TRUE (represented in the C programming language by the integer 1) is returned to the Strategic level and bound to the variable ANS1. The primitive goal is then considered to be satisfied. Next, the return value is checked. If TRUE was returned, the comparison "ANS1 == 1" succeeds and the next primitive goal, "select_first_waypoint(ANS2)", is reached. If, however, the value of ANS1 had been FALSE due to some failure encountered in readying the vehicle, the comparison would fail, causing the first "initialize" rule to fail. The inference engine would then select the second "initialize" rule. This rule involves a primitive goal commanding the Tactical level to alert the user, followed by a Prolog *fail* predicate that forces the failure of the rule. Since no other "initialize" rules exist, the "ini-

tialize" subgoal of the "execute_auv_mission" rule fails and, thus, so does the original query, representing failure of the entire mission.

Assuming the initialization process succeeds, the Tactical level is then commanded to select the first waypoint. Once this behavior has been performed, the first "initialize" rule succeeds. Prolog then returns to the "execute_auv_mission" rule and proceeds to the right in its attempt to satisfy all the remaining subgoals sequentially. The *repeat* predicate is encountered and succeeds automatically. This identifies the entry into the logical control loop consisting of the subgoal "mission". There are four "mission" rules in the rule base, one for each phase: transit, search, task, and return. These rules are ordered so that the satisfaction of each will occur in the desired sequence.

The first "mission" rule implements behaviors supporting the transit to the target area. First, a query "in_transit_p(ANS1)" is sent to the Tactical level to verify that the mission is truly in the transit phase. Although this may appear a bit unusual, it is necessary because the Strategic level cannot maintain an explicit state of its own and must rely on the Tactical level for status information related to the vehicle, the mission, and the environment[2]. The response to the "in_transit_p" query is returned and bound to the variable ANS1[3], a comparison is done, and, if the mission is in the transit phase, the subgoal "transit" is searched.

Two "transit" rules are in the rule set. The first rule contains the "waypoint_control" subgoal, and the other contains the "surface" primitive goal. The "waypoint_control" subgoal is further decomposed, as specified by the rule of that name, into a series of "critical_system_prob" checks, a waypoint status check, a planning sequence (if necessary), and a directive to issue new setpoint and mode commands to the Execution level.

---

2. These restrictions result in Prolog code devoid of assert and retract statements, unification, and quantification. The use of variables is allowed only in conjunction with primitive goals (foreign function calls) and even then may only be of type Boolean.

3. Similar variable names shared among different rules implies nothing. In Prolog, the scope of a variable extends only within the rule in which it resides. Therefore, when the search moves to a different rule, previously bound variables are not accessible.

One "critical_system_prob" rule is included in the rule set for each vehicle subsystem which could, upon failure, jeopardize the mission. These rules, logically OR-ed together, each involve a query to the Tactical level relating to the status of one such subsystem. These systems are checked, in sequence, every time through the mission control loop. If one system is reported to have failed, the corresponding "critical_system_prob" rule will succeed. Returning to the "waypoint_control" rule, the truth of the subgoal "critical_system_prob" is negated by the *not* predicate, a system-provided logical operator. Hence, a critical system failure will cause the failure of the "waypoint_control" rule which in turn causes the "transit" rule to fail. However, if each "critical_system_prob" rule fails, the failure is negated within the "waypoint_control" rule causing the search to proceed to the "get_waypoint_status" subgoal. This subgoal consists of the logic needed to determine whether a Global Positioning System (GPS) reading is needed and, if it is, to obtain it. Next, the Tactical level is asked if the current waypoint has been reached. If so, a command to select the next waypoint is issued. The facts "gps_needed" and "get_waypoint_status" provide for success by default for their respective subgoals. In other words, if the corresponding rules fail, then the subgoal will succeed when these facts are reached.

The "plan" rules provide the logic for replanning the mission. The first plan involves *global replanning* as a result of reduced system capabilities. In this scenario, non-critical vehicle systems are checked which, if faulty, would not necessarily threaten the success of the mission. Instead, these faults would require a replan that may possibly result in a truncated mission or the relaxing of certain performance requirements. If global replanning is not needed, then the next "plan" rule checks whether an uncharted obstacle has been identified. If so, a *local replan* is performed. Note that the rules for both global and local replanning contain commands to initiate a "loiter" routine prior to performing the replan. By placing the vehicle in this mode, the Tactical level is free to concentrate its resources on the problem of replanning. It is conceivable, and probably desirable, to move some of the decision-making responsibilities of replanning to the Strategic level. The level of logical sophistication is determined by the mission specialist and probably depends on the com-

plexity of the replanning schemes to be used. If neither type of replanning is required, a "plan" fact is included to represent the nominal operating condition and to insure the success of the "plan" subgoal.

Once the "plan" subgoal has been satisfied, one last subgoal of the "waypoint_control" rule must be satisfied. This subgoal is the primitive goal "send_setpoints_and_modes(ANS)" and whose intent is to direct the Tactical level to issue the new command packet to the Execution level. After the packet is sent, the Tactical level returns acknowledgment to the Strategic level, completing the evaluation of the "waypoint_control" rule.

Should the first "transit" rule fail due to the failure of the "waypoint_control" subgoal, the second rule will activate a behavior causing the AUV to surface. Surfacing is performed only as a last resort, but the inclusion of this rule is necessary to avoid an exceptional condition for which the vehicle has no response. If, for instance, the first "transit" rule failed due to a systemic fault, and a second rule initiating the surface routine wasn't available, the Strategic level would be "stuck" between the failure of the transit subgoal and the success of the *repeat* construct in the "execute_auv_mission" rule, producing an unacceptable infinite loop.

Given that the various vehicle subsystems are healthy and a replan is not required, the first "transit" rule will succeed. Prolog's inference engine will return to the originating "mission" rule and continue its attempt to satisfy the rule. When moving from left to right, the cut succeeds, and the next subgoal is investigated. Depending on the response to the primitive goal "transit_done_p" and the subsequent comparison, Prolog will either reach the *fail* predicate at the end of the rule or backtrack to the last subgoal. In the former case, the transit phase has reached completion and the *fail* forces failure of the first "mission" rule. Prolog then searches for an alternative path to satisfy the "mission" subgoal. In the latter case, the "transit" phase has not yet completed, and backtracking commences from the comparison "ANS2 == 1". Primitive goals, in their capacity as function calls, fail during backtracking and are passed over; therefore, "transit_done_p" is not retried. This

leads to an encounter with the cut. For the purpose of the search, this predicate identifies the point from which remaining subgoals are not to be reconsidered. In effect, the rest of the rule body is "cut" away, and Prolog will drop its attempt to satisfy the "mission" subgoal. Backtracking then returns to the "execute_auv_mission" rule where the *repeat* causes Prolog to again attempt to satisfy the "mission" subgoal.

The search, task, and return phases of the mission are completed in much the same way. Mission rules for each are structured similarly, in that a verification of the phase is made, followed by the logic relevant to that phase, and a check for phase completion. At the Tactical level, flags are maintained identifying the current state of the mission. The Tactical level is also responsible for recognizing the end of each phase. When a phase is completed, the flag associated with that phase is reset and the flag for the next phase is set. As in the case of the transit phase, the other phases have two rules associated with them: one representing the normal case and one meant to deal with unrecoverable failures. For the purpose of this evaluation, any anomalies of this type result in the decision to surface. This alternative rule is always turned to as a last resort and, hence, is placed after its normal-case counterpart.

When the return phase of the mission is completed, signified by a TRUE response to the "return_done_p" subgoal of the fourth "mission" rule, the primitive goal "wait_for_recovery(ANS3)" is reached. This goal directs the Tactical level to place the vehicle in a state compatible with recovery. After this has been accomplished, the fourth "mission" rule succeeds, satisfying the "mission" subgoal of the "execute_auv_mission" rule. Since all of its subgoals have been satisfied, a *yes* response is given by Prolog to the original query "?-execute_auv_mission.", indicating successful mission completion.

The mission specialist responsible for programming the mission determines to what degree control is to be exercised from the Strategic level and the amount of responsibility to be delegated to the Tactical level. The primitive goals "do_search_pattern", "start_local_replanner", and "start_global_replanner" are prime candidates for further decomposition at the Strategic level. In addition, seemingly straightforward goals such as

"surface" may need to be more precisely specified to account for circumstances in which the vehicle could behave differently to achieve the same goal (i.e., driving to the surface versus blowing ballast tanks).

The clauses of the Prolog program in Appendix A-1 have been divided into two groups marked "Mission Specification" and "NPS AUV Doctrine". The purpose of this is to identify those rules which can vary from mission to mission (the Specification) and those of a more general nature which would be applicable for any mission (the Doctrine). This partition is purely to assist in human understanding. The inference engine views the entire program as a single rule set and searches it accordingly.

b. The Forward Chaining Implementation

The original goal-driven solution to the search-and-rescue mission may also be implemented using a forward-chaining approach. The resulting control architecture is an instantiation of the Rational Behavior Model with a forward-chaining Strategic level (RBM-F). As can be seen from the source code listing of Appendix A-2, a great deal more complexity is involved as compared to the backward-chaining example. The reason for this is twofold: (1) forward chaining rules are better suited to data-driven as opposed to goal-driven problems and (2) forward chaining solutions rely on the explicit identification and maintenance of the problem state space, tasks that were handled implicitly by the inference engine in the Prolog implementation.

Regardless of the direction of the chaining selected for the Strategic level, the sequence of primitive goals attained during mission execution must be identical to that specified by the goal-driven solution. In terms of a forward-chaining system, this implies a particular sequence of state transitions. A State Transition Diagram is a graphical representation of the relevant states of a problem and the relationships (or transitions) between them.

In a forward chaining implementation, the rules represent the transitions. Each rule consists of two parts: a condition (or antecedent) and an action (or consequent),

similar to an IF-THEN statement in an imperative language. If the condition side of a rule is satisfied by the current state of the problem, then the rule is said to be active. Should the actions specified by the consequent side of the active rule are applied, the rule is said to be fired.

Except for final or "trap" states, each state has one or more successor states associated with it. If a single transition exists, the corresponding rule is fired which alters the facts (state) accordingly. When multiple transitions lead from the current state, the potential exists for a *conflict*. This occurs when the rules defining the transitions are simultaneously active, and each must be investigated to determine which is to be selected for firing. The resolution of these conflicts is generally the responsibility of an arbitrating entity (such as an inference engine).

For this work, CLIPS (an acronym for C-Language Integrated Production System [Ref. 134]) was selected. CLIPS is a forward-chaining, rule-based expert system shell which mimics both LISP and OPS5, two languages whose features form the foundation of CLIPS. Another important characteristic that CLIPS shares with its predecessors is its use of the Rete pattern-matching and inference algorithm [Ref. 185]. The form of a CLIPS rule is shown in Figure 9. The antecedent of the rule lies on the left-hand side of the

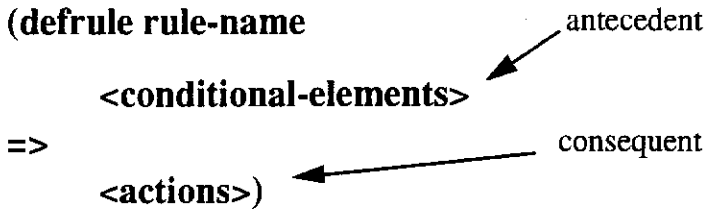


Figure 9. A CLIPS Rule

production arrow (=>). The remaining portion of the rule resides on the right hand side of the arrow and is referred to as the consequent. The antecedent is a set of conditions which must be satisfied for the rule to be applicable. In CLIPS, the conditions of a rule are satisfied

based on the specified facts in the database (called the fact-list). The CLIPS inference engine matches the various conditions (patterns) against the current fact-list and determines which rules are applicable. Rules whose left-hand sides are matched by the facts are placed on the list of active rules called the *agenda*. CLIPS provides a full complement of logical connectives to relate multiple conditional elements.

The consequent of a rule is a set of zero or more actions that are applied when the rule is executed. Rule firing, as well as the resolution of conflicts between active rules, is done by the inference engine. Whereas Prolog relied on depth-first search as its sole conflict resolution strategy, the CLIPS system provides several possible strategies. Of particular significance in this work is the facility provided by CLIPS to assign *saliences* (priorities) to the rules. In any case, the actions of the executed rule are performed, typically causing a change of state in the form of retracted and asserted facts. The new fact-list may in turn affect the list of applicable rules, causing new rules to be activated while other previously active rules are removed from the agenda. This cycle of rule selection and execution continues until no applicable rules remain.

Although similar in many ways to the procedural IF-THEN statements of imperative languages, CLIPS rules are selected and remain active as long as their conditions are satisfied. In this way, any number of rules may be available for execution at a given point during problem solution. It is the responsibility of the inference engine to insure that the list of applicable rules is always kept current.

Within CLIPS, states are defined by sets of facts contained in the current fact-list. These facts may be structured using the CLIPS *deftemplate* construct. The resulting fact template defines a group of related fields into which facts are placed. Each field may have associated with it an explicit type definition, a finite list of allowable values which the field can accommodate, and the default value of the field.

Appendix A-2 contains the forward chaining Strategic level code for the search and rescue mission written in CLIPS. The template definitions, which have been gathered together at the front of the program solely for ease of understanding, define the

relevant states of the goal-driven search and rescue mission solution. The goal-driven solution is the same one used to develop the backward-chaining implementation. However, in order to preserve the same sequence of behavior activations (or, equivalently, the same ordering of primitive goals), additional states reflecting alternative solution paths had to be defined. These alternative paths are enumerated in the "allowed-symbol" slot of the corresponding deftemplate. These templates are identified, in turn, by the inclusion of the prefix "or" in the label of the deftemplate structures. In affect, these templates provide the "traffic control" necessary to resolve conflicts between simultaneously active transition paths.

The remaining CLIPS code for this implementation comprises the rule definitions which are identified by the *defrule* keyword. Each rule supports the attainment of some goal defined in the AND-OR goal tree produced by the goal-directed problem decomposition of the search-and-rescue mission. To assist in understanding, a particular labeling convention was used for these rules. As an example, the rule "waypoint-control-13-s" is analyzed here. The name "waypoint-control" is derived directly from the goal of the same name. The first digit of the numerical suffix refers to the first (highest priority) path to satisfaction of this goal. In the case where alternative paths exist, this digit would range from 1 to the total number of paths. In Prolog, these alternatives are manifested in multiple rules which share identical left-hand sides. The second digit of the numerical suffix represents the target subgoal (state) whose satisfaction is currently being attempted. Continuing the example, the suffix 13 would refer to the transition to the third state ("plan") of the first inference path of "waypoint-control" (of which there is only one). The letter "s" (for *start*) indicates that this rule involves the transition *into* the plan state; a letter "e" (signifying *exit*) would identify a rule whose firing would assert facts indicating the success of the goal associated with that state.

In the Prolog solution, a response corresponding to failure caused the inference engine to halt its attempt to satisfy the goal specified by the head of the current rule. The search path would then proceed, in depth-first fashion, to the next rule containing an identically labeled head. By sharing the same rule left-side, these rules represented alterna-

tive (ORed) solutions. In addition, the priority of these rules with respect to each other was determined by their placement in the rule set.

An analogous prioritizing of OR-related rules does not exist in CLIPS. Textual ordering of the rules is irrelevant; instead simultaneously activated rules are ordered based on their saliences. When a conflict occurs due to the existence of multiple state transition paths, the transition (rule) with the highest priority is taken. Each subsequent rule firing along that path may include a primitive goal whose purpose is to query the Tactical level about the state of the mission or vehicle. The response to each query could potentially lead to the failure of this reasoning path. This in turn requires that the state of the OR-relation be restored, allowing the alternative path(s) to be investigated. Restoration of these states is the purpose of the "traffic control" slot values within deftemplates containing the word "or" in their designators.

Once the CLIPS environment is entered, the "(start)" fact is asserted onto the fact-list. The CLIPS inference engine searches all the rule left-hand side patterns to identify active rules which, when found, are placed on the agenda. In this example, the rule "execute-auv-mission-10-s" is activated because its condition is satisfied. Since it is the only rule on the agenda, it is fired, resulting in the retraction of the "(start)" fact and the assertion of the fact "(execute-auv-mission (state initialize))". This fact represents the state which corresponds to the "initialize" subgoal of the execute_auv_mission Prolog rule. The CLIPS inference engine then searches the rule set, attempting to match this new fact against the condition patterns of the rules. The rule "execute-auv-mission-11-s" is then activated and it is fired. This results in the assertion of the fact "(initialize (state start))". Again, the inference engine searches the rule set, and the rule "initialize-10-s" is activated and fired. The firing of this rule leads to the first of two alternative branches specified by the "or-initialize" template with its states "or_initialize_1" and "or_initialize_2".

At this point, the fact-list holds the two facts "(or-initialize (state or_initialize_1))" and "(initialize (state select_first_waypoint))". The rule "initialize-11" requires these two patterns to satisfy its left-hand side, in addition to the primitive goal "ready_ve-

hicle_for_launch". In order to determine if this condition is applicable, this query must be answered by the Tactical level. The response is then used in the "(test (= (ready_vehicle_-for_launch) 1))" expression. This involves the invoking of the function "ready_vehicle_-for_launch" (the same used in the Prolog implementation) which returns a Boolean value. The reserved word "test", along with the "=" function, performs an equivalence comparison between this response and the number 1. If the comparison succeeds, the rule "initialize-11" will fire. If, however, the comparison fails, the rule "initialize-11" will not be activated; instead, the rule "initialize-10-e" is fired. This rule retracts the "(or-initialize (state or_initialize_1))" fact, breaking this inference chain, and establishing the alternative path by asserting the fact "(or-initialize (state or_initialize_2))". This fact will lead to rules causing the primitive goal "alert_user" to be attained.

Presuming that the initialization phase of the mission is successful, the mission rules are then considered. These rules correspond to the four phases of the search-and-rescue mission: transit, search, task, and return. In a manner similar to that describing the initialize rules, these rules are activated and fired based on saliences, attainment of a goal, and the results of queries. The functionality of the special Prolog control constructs *repeat*, *!*, and *fail* are emulated through the formulation of the left-hand side of each rule.

To verify that the forward-chaining implementation did in fact produce the same side effects as the backward-chaining version, a trace of the sequence of primitive goals reached by each was collected for examination. The two versions of RBM did produce identical sequences of behavior activations. The trace for a completed search-and-rescue mission is found in Appendix A-3. Also included is a trace of an experiment involving the failure of the power supply of the vehicle. Traces such as these can be used to verify the mission logic upon which the Strategic level is instantiated.

2. Tactical Level

The Tactical level developed for this study is shown in Figure 10. This design was patterned after the watch crew of a manned submarine [Ref. 186], because the division of

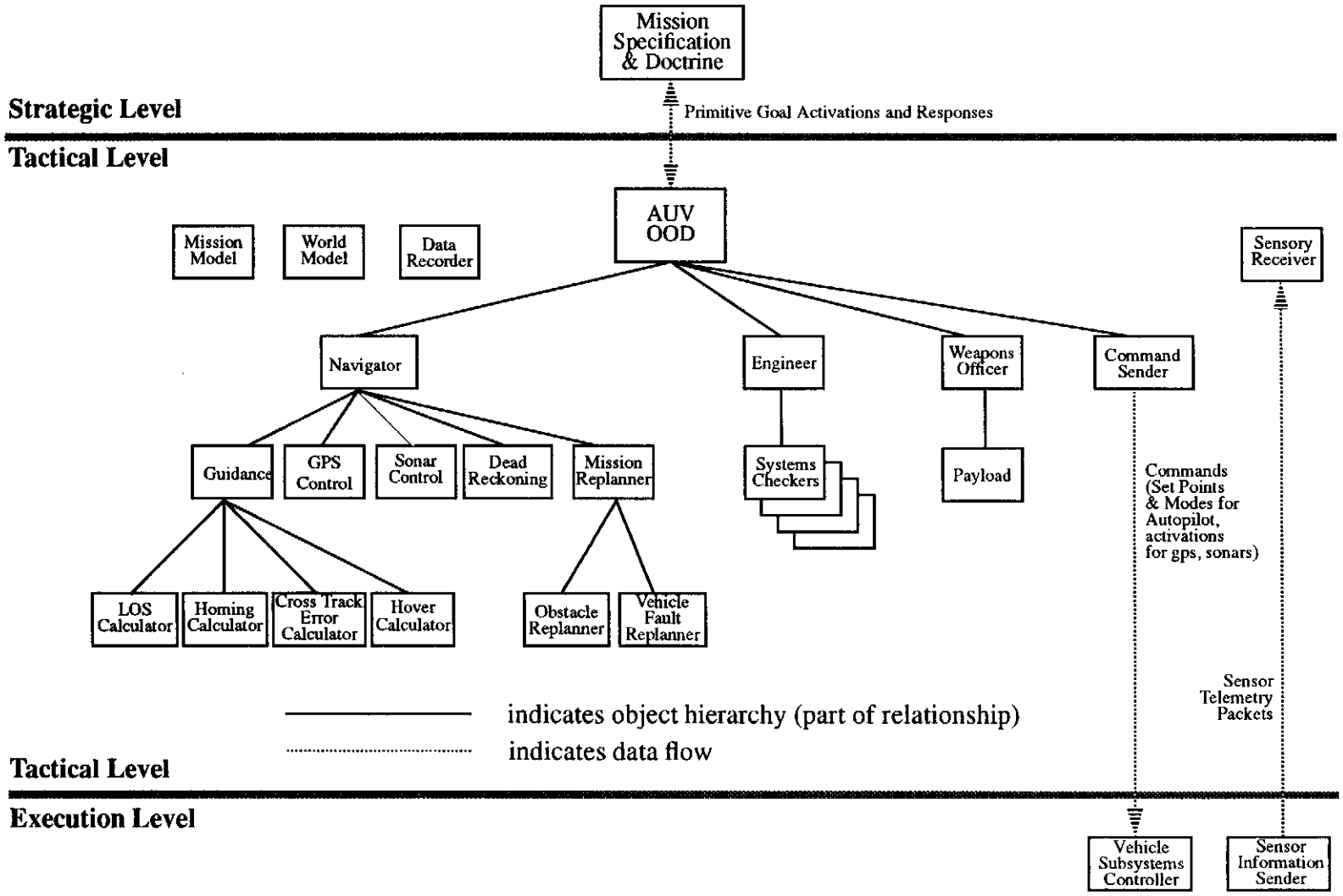


Figure 10. Tactical Level Objects for the NPS AUV Florida Mission

responsibility and command relationships between parties was already well understood. Each block in the diagram represents a distinct entity within the organization and corresponds to a software object. Most of the objects are arranged into a hierarchy as indicated by the dotted lines linking them together. The AUV Officer of the Deck (OOD) resides at the top of the hierarchy and assumes overall control of the operation. In addition, the OOD provides the single interface between the Strategic and Tactical levels. Primitive goals from the top level are passed to the OOD who in turn activates behaviors known to the Tactical level and designed to satisfy those goals. Using the analogy of the watch crew, the Captain of the submarine issues commands to or asks for status reports from the OOD. The OOD then turns to his subordinates and issues the appropriate orders to satisfy the goal or query presented by the Captain.

All the behaviors that are capable of being performed by the vehicle are embodied within the various objects of the Tactical level. The OOD must coordinate the actions of each object to insure each task is accomplished as expected. The behaviors, for their part, are reflected in the methods contained within the applicable object(s). Typically, a behavior will involve the interaction of multiple objects. Necessary inter-object communications are provided through the passing of messages. As depicted in Figure 10, direct communications between members of the hierarchy is restricted to parent-child links. While this comes at the expense of efficiency, the benefits include the avoidance of unconstrained communication paths and a greater degree of modularity. These characteristics support RBM's emphasis on providing a framework to the user that aids in the understanding and maintenance of the software at this level.

Communications with the Execution level is likewise restricted. Commands, in the form of packets containing numerical set points and discrete mode changes, are issued only from the Command Sender object under the supervision of the OOD. Likewise, telemetry data from the Execution level is received solely by the Sensory Receiver object. By constraining these interfaces, the potential for command conflict and data inconsistency is avoided.

Several objects at the Tactical level are not explicitly connected to the object hierarchy. These represent databases or data stores intended to be accessed by any requesting objects. The state of the mission, the environmental model, current sensor readings, and mission history are maintained and encapsulated within the corresponding objects. Requests for information and data updates are handled as they arrive; however, these objects do not participate in task accomplishment directly.

The choice of an implementation language for this design was made from a range of object-oriented and object-based programming languages. The candidate languages considered included Ada, C++, and CLOS. Each provide constructs for data abstraction, information hiding, instantiation of objects, and, with one exception, inheritance. The exception is Ada, which does not support the concept of class hierarchies. However, Ada does provide the control primitives needed for concurrency which are not found in the other languages. This potential for parallelism, coupled with the availability of an object-oriented extension to Ada called Classic-Ada [Ref. 92], were the deciding factors in its selection as the language for the Tactical level.

A brief description of each object is now given. This description will be limited to the purpose of the object as well as any significant or unusual characteristic which may be of interest to the reader. The source code use in this implementation, including the data structures, methods, and interface specification of each class can be found in the Classic-Ada source listing included in Appendix B.

For the purpose of the evaluation that follows, the complete object hierarchy was instantiated. Not all behaviors, however, were fully implemented. Details of the individual behaviors (e.g. path planning and fault recovery) are not of primary interest here. Instead, the integration of these numerous and diverse behaviors is one of the fundamental requirements of any software architecture and, hence, will be the focus of this investigation.

a. AUV OOD

The OOD provides the sole interface between the Strategic and Tactical levels. Thus, this object, upon receipt of a primitive goal, must identify and activate the corresponding behavior(s) needed to satisfy that goal. Some behaviors may require the activation of several objects. These activities, whether performed sequentially or concurrent with each other, must be coordinated by the OOD.

The OOD has four immediate subordinates: the Navigator, the Engineer, the Weapons Officer, and the Command Sender. It is to these objects that the responsibility for supporting goal achievement is delegated. If these objects, or the dependents of these objects, require services from a sibling or dependent of a sibling, that request, in the form of a message, must be sent through the object hierarchy to the OOD. The OOD must then route the message to the appropriate subordinate.

Commands to the Execution level are composed of a mode and setpoints specifying the desired heading, depth, and speed. These components are calculated by different objects after which they are passed to the OOD. There, they are assembled into a single command packet and sent to the Command Sender for subsequent passage to the Execution level.

b. Navigator

The Navigator is primarily responsible for the current and future location of the vehicle. Hence, the tasks of guidance, position estimation, and path replanning fall under the purview of this object. Like the OOD, the Navigator is responsible for the dissemination of orders to its subordinates and the coordination of their actions.

c. Guidance

This object provides the desired heading setpoint which is eventually included in the command to the Execution level. This value may be obtained through the application of a *guidance law*, an algorithm that accepts parameters relating to current position, desired position, and environmental dynamics and produces a value corresponding to the

angle in which the vehicle needs to align itself. Several algorithms may be implemented, including line-of-sight (LOS), Cross Track Error, and variations for homing and hovering.

For this research, a LOS guidance calculation was employed. When tasked to provide a new heading, this object performs the following calculation:

$$\Psi cmd = \text{atan}\left[\frac{(Ynext - \hat{Y})}{(Xnext - \hat{X})}\right]$$

where $(\hat{X}, \hat{Y})$ is the AUV's current location, (*Xnext, Ynext*) is the next waypoint to be achieved, and Ψcmd is the angle between the reference and the line joining the two points. The result, in radians, is then converted to degrees, as expected by the steering autopilot.

d. GPS Control

The methods to activate, monitor, and take readings from the Global Positioning System receiver are contained here. This system provides the ability to locate the position of the vehicle to a high degree of accuracy [Ref. 187]. Although this system is scheduled to be implemented on the NPS AUV, its functionality was not included in this simulation.

e. Sonar Control

This object is responsible for the issuance of sonar commands, object identification and classification, and estimating vehicle location based on this interpretation. This capability was not modeled for this study. Related research includes [Ref. 188].

f. Dead Reckoning

The purpose of this object is to provide an estimate of the vehicle's position based on a known position fix and the elapsed time since that fix. In the NPS AUV, dead reckoning output is required by the synchronous processes; hence, a separate dead reckoner is implemented in the Execution level. Although not essential, the Tactical level dead reck-

oner may be used as a way of checking the consistency and validity of Execution level operation.

g. Mission Replanner

Two types of replanning is performed by this object: local replanning caused by an uncharted obstacle and global replanning driven by a vehicle fault. Neither was modeled for this study.

h. Engineer

The Engineer monitors the health of the vehicle. Systems and conditions of interest include power, computer, propulsion, steering, diving, buoyancy, thrusters, pressure, and temperature. For this study, only the power system was modeled.

i. Weapons Officer

The primary concern of the Weapon's Officer is the viability and delivery of the vehicle's payload. No payload was modeled for this study.

j. Command Sender

This object accepts command packets from the OOD and transmits them to the Execution level. Since the Tactical and Execution levels are physically separated in this instantiation, the Command Sender incorporated the software necessary to perform network communications. Each command packet consists of the following: heading command, depth command, speed command, and mode. Commands pertaining to latitudinal and longitudinal positioning are also sent; however, these relate to the AUV's hover mode which was not used in these experiments.

k. Sensory Receiver

This object accepts telemetry records from the Execution level. Subsequently, it extracts and stores individual data values and makes them available to other objects in the Tactical level. This object also affixes a time stamp on each sensory packet before sending the packet to the Data Recorder for archival purposes. In this instantiation, the Sen-

sory Receiver was synchronized with the Execution level at a 2 Hz transfer rate. Each sensory packet consists of a position given as (X, Y, ALT, depth) where X and Y are cartesian coordinates mapped to the NPS swimming pool, ALT is the distance above the floor pool, and depth is the current depth of the vehicle [Ref. 189].

l. Mission Model

This object contains the list of waypoints generated off-line by the mission planner, including the vehicle's start and finish location. It also maintains the flags indicating the current phase of the mission. For the purpose of this dissertation, methods and data structures are also included to support the initialization of the simulator.

m. World Model

This object contains maps, lists of known objects, and other data reflecting the vehicle's operational environment that are available prior to mission initiation.

n. Data Recorder

This object maintains all telemetry records passed to it by the Sensory Receiver, plus any additional messages designed to explain unforeseen or unusual events. Data is configured to best support post-mission analysis. Further details concerning these issues, as well as the communication interface incorporated within the Data Recorder, may be found in [Ref. 190].

3. Execution Level[4]

The instantiation of this level is patterned after the single loop controller implemented in the actual vehicle. Once the communications have been established and the simulation initialized, the program runs in a continuous loop broken only by periodic data exchanges with the Tactical level. Each pass through the loop consists of determining the depth of the water under the keel, checking the system clock for the next exchange with the

4. The software for this level was written by S. M. Ong [Ref. 189], D. B. Nordman [Ref. 184], and D. Marco [Ref. 191].

Tactical level, an additional clock measurement in support of the real-time graphics update, calculating vehicle dynamics and position based on the time since the last frame was drawn, and finally a redraw of the simulation frame. Communication with the Tactical level consists of the receipt of commands followed by transmission of telemetry. The target exchange rate on the actual AUV is 10 Hz. However, due to the time required by the graphics portion of the program, a 2 Hz communications rate was realized in the simulation study of this dissertation.

To provide realism to the simulation, the mass characteristics of the NPS AUV were modeled along with hydrodynamic coefficients for longitudinal, lateral, and normal forces as well as roll, pitch, and yaw. The vehicle model program calculates the dynamic equations of motion using readings from rudders, dive planes, and propeller rpm. The output is X, Y, and Z in world coordinates and the vehicle azimuth, elevation, and roll Euler angles. Drag force is calculated and integrated over the vehicle using four-term gauss quadrature. The resulting NPS AUV simulation is shown in Figure 11.

E. EXPERIMENTS

Complete instantiations of both the Rational Behavior Model-Backward and Rational Behavior Model-Forward were implemented. The Strategic level of the architecture, written in Prolog (for RBM-B) and CLIPS (for RBM-F), ran on a Sun SPARCstation 1 under the UNIX operating system. The interpreted rule set, derived from a top-down goal decomposition of the Florida search and rescue mission, is included for each form of RBM in Appendix A. The Tactical level was written in Classic-Ada, an object-oriented extension of Ada. The Classic-Ada compiler produces "pure" Ada source code which is ready for compilation. Verdix Ada running under UNIX on a second Sun SPARCstation 1 was used for the generation of the executable modules. The Classic Ada source code listings are located in Appendix B. The simulation, implemented in the C programming language, ran on a Silicon Graphics 4D/240VGX workstation under the IRIX operating system. The three workstations were linked over an Ethernet connection. All communications was accomplished

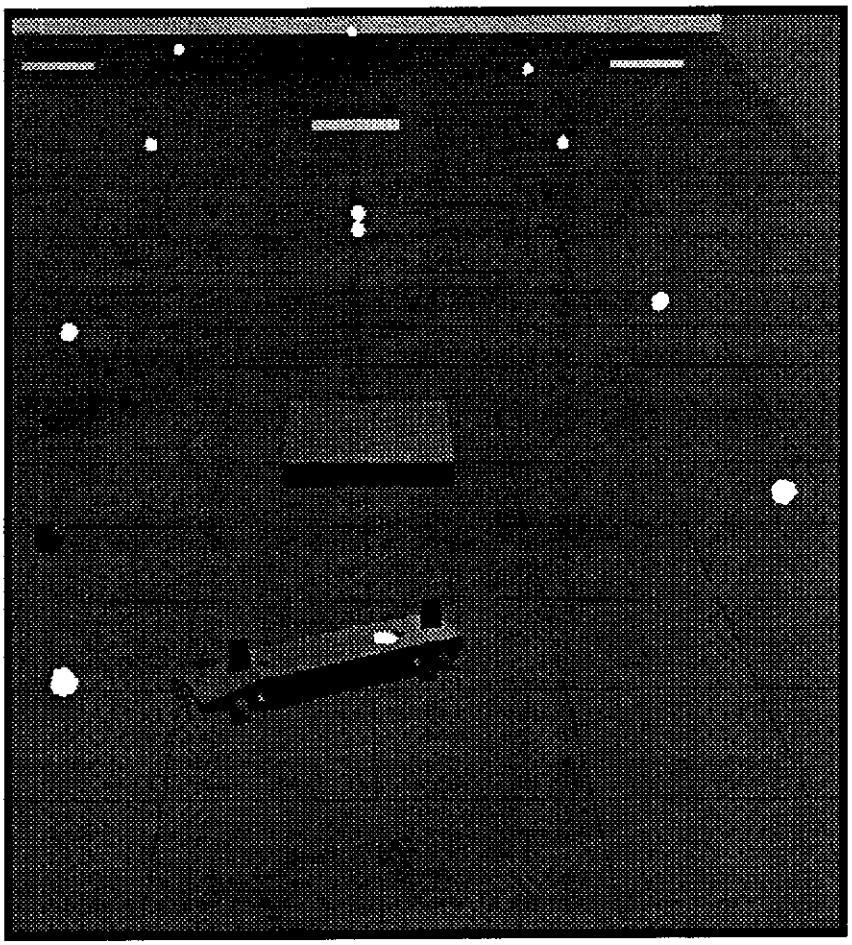


Figure 11. The NPS AUV Simulation

using stream sockets. Data values were converted into ASCII characters prior to transmission and reconverted to their original type following receipt. The data was transmitted in a form whereby the data type and size was explicitly included. The software used to provide the communications support was written in the C programming language by Professor S. H. Kwak and is included at Appendix C.

The first experiment involved the traversal of a figure-8 around the NPS pool. In order to better represent the four-phase Florida mission, the primitive goals "search" and "task" were assumed to be accomplished at the half-way point. The "traversal" and "return" phases were represented by movement along the respective halves of the path. The results of a

typical time for the trial controlled by Prolog and CLIPS versions of RBM are compiled in Tables 3 and 4, respectively.

Table 3: PERFORMANCE OF RBM-B IN TRANSIT PHASE

Waypoint	Time/Total (seconds)	X position (in)	Y position (in)	Depth (in)	Heading (degrees)
(350,100,10)	102/102	332.93	92.73	9.61	71
(500,250,30)	66/168	488.78	235.58	28.75	42
(500,500,35)	77/245	500.63	480.77	33.82	355
(350,700,40)	76/321	363.38	686.14	38.53	319
(200,900,40)	71/392	211.95	885.78	38.40	321
(200,1150,12)	78/470	197.18	1130.69	13.36	7
(350,1300,20)	66/536	335.49	1287.62	18.81	50
(500,1150,30)	81/617	491.04	1167.50	28.88	154
(500,900,40)	77/694	498.18	918.40	38.08	178
(350,700,50)	77/771	361.38	715.22	48.17	219
(200,500,50)	71/842	209.00	516.68	47.91	211
(250,150,50)	104/946	244.66	169.22	50.35	166

Associated with each waypoint is the time required by the AUV to reach it since the last waypoint, along with the total elapsed time of the run and the actual position and heading of the vehicle when waypoint attainment was determined. A waypoint was considered reached when the pythagorean distance between it and the center of mass of the vehicle was less than 20 inches. Although calculated using only two dimensions (X and Y), the vehicle was equally adept at maintaining the desired depth.

The close similarity of performance between backward and forward chaining implementations of the RBM indicate that both Strategic levels performed sufficiently to satisfy the 2 Hz update rate required by the Execution level. This implies that the complexity of

Table 4: PERFORMANCE OF RBM-F IN TRANSIT PHASE

Waypoint	Time/Total (seconds)	X position (in)	Y position (in)	Depth (in)	Heading (degrees)
(350,100,10)	104/103	332.97	90.92	11.89	63
(500,250,30)	66/169	488.76	235.36	28.63	41
(500,500,35)	78/247	499.31	480.06	33.73	359
(350,700,40)	77/324	362.75	685.41	38.53	318
(200,900,40)	71/395	211.62	884.92	38.46	323
(200,1150,12)	78/473	200.60	1131.26	13.50	2
(350,1300,20)	69/542	334.06	1290.34	18.69	57
(500,1150,30)	78/620	490.85	1167.41	28.78	152
(500,900,40)	77/697	500.46	919.38	38.19	182
(350,700,50)	76/773	363.94	714.32	48.01	222
(200,500,50)	71/844	210.70	516.42	47.77	217
(250,150,50)	104/948	246.73	169.28	50.53	167

each Strategic level may increase up to the point at which the time taken by the Strategic level to reason about the next goal exceeds the minimum acceptable update rate between the Tactical and Strategic levels.

To examine the impact a different computational configuration would have on the performance of RBM, a two workstation test was performed. In this case, the Strategic and Tactical levels shared the processor of a single Sun SPARCstation 1. Again, UNIX was the operating system and communications between the Tactical and Execution levels took place over Ethernet. The operating system was relied upon to provide context switching between the Strategic and Tactical levels. Not surprisingly, the results of this experiment were essentially identical with those of the three-workstation configuration for both RBM-B and RBM-F. The reason for this lies in the relationship between the two levels in this implemen-

tation. When the Strategic level is searching for the next primitive goal, the Tactical level is dormant. Similarly, after the primitive goal has been issued, the Strategic level sleeps until it is notified by the Tactical level of goal attainment. Note that this instantiation did not implement any active objects. If it had, the two-workstation configuration would have suffered in performance because the active objects would have been blocked each time the Strategic level was given access to the processor. In a scenario in which the Tactical level includes active objects, a three-workstation configuration would allow the Tactical level continuous access to a processor. Of course, the objects themselves would be required to compete for the single processor; nevertheless, with the Strategic level searching its rule base, true parallel execution could then occur between all three levels of RBM.

A trace of the primitive goals during this trial is included in Appendix A-3. This trace begins with the activation of the inference mechanism followed by a successful initialization and the selection of the first waypoint. The mission then moves into the transit phase. The primitive goals supporting this phase are presented in a continuous sequence to the Tactical level which in turn activates the associated behaviors. During each "cycle", the Strategic level checks for waypoint attainment and directs the selection of the next waypoint, if appropriate. As can be seen, the majority of the goals supported the transit and return phases of the mission. This is due to the general nature of the "search" and "task" goal. The expansion of the logic associated with either of these would by necessity involve additional primitive goals.

The next experiment was designed to test the ability of the Strategic level to reason in the face of a subsystem failure. A battery model was included whose voltage level was a function of time. Again, the resulting trace is given in Appendix A-3. As expected, when the voltage falls below a specified threshold, the "transit" rule corresponding to the nominal case fails. The search them moves to the second rule, included to account for failures of this type. The trace indicates that the "surface" goal was encountered, and performance of this behavior was verified visually on the simulator.

F. PERFORMANCE DISCUSSION AND EVALUATION

The importance of "visual verification" should not be taken lightly in the domain of autonomous vehicle control. Evaluation of a control software architecture cannot be subject to mathematical proof due to the complexity and diversity of the software involved. Furthermore, the concept of "correctness" as it relates to an autonomous vehicle's behavior is a subjective one. One may speak of a vehicle's "robust behavior in an unstructured environment" but be unable to devise proofs supporting these observations due to the lack of precise definitions [Ref. 62]. In lieu of this situation, evaluation must be based on actual performance [Ref. 192].

Nevertheless, some amount of verification and validation of software should occur prior to installation on the actual vehicle. A trace of the logic embodied in the rule set of the Strategic level should be examined under a variety of circumstances to insure the proper sequencing of primitive goals and, by extension, behaviors. Some behaviors, particularly those performing numeric or iterative computation, may be candidates for algorithmic analysis. Individually, these can be shown to be correct from a mathematical standpoint. Taken collectively, this analysis should include the resulting commands to the Execution level to determine if goals are being effectively translated into vehicle actions.

The ultimate test, however, is the demonstration of satisfactory performance on the target vehicle. This, of course, can only be determined qualitatively by the individual whose opinion carries the most significance--the user.

G. SUMMARY

This chapter presents a complete instantiation of the Rational Behavior Model using different programming paradigms for each level. Specifically, the Strategic level was implemented in Prolog, the Tactical level in Ada, and the Execution level in C. To demonstrate the effectiveness of this architecture, a study was performed involving a realistic simulation of the NPS AUV. A four-phase search and rescue mission formed the basis for the logic embodied in the rules of the Strategic level. The Tactical level design was patterned after the

watch crew of a manned submarine, partly because of the clear division of responsibility and chain of command.

The experiments were performed on both two and three workstation configurations. The motivation for this was to demonstrate the feasibility of a heterogenous approach to the implementation of software architectures. Traces were taken of the execution of the Strategic level inference engine, and timing, positional, and heading commands were collected as output from the Tactical level. However, it was the observed performance of the AUV simulation that determined the success of the RBM architecture as a framework for the control of an autonomous vehicle.

VII. SUMMARY AND CONCLUSIONS

In this dissertation, a complete, tri-level, multi-paradigm software architecture for the control of autonomous vehicles is defined, instantiated, and evaluated. By definition, each level of the Rational Behavior Model has associated with it a particular abstraction mechanism which, when applied to the control problem, yields a solution that emphasizes software maintainability, modifiability, and a potential for reuse. At the Strategic level, high-level mission logic resides which provides for the deterministic sequencing of the underlying behaviors of the vehicle. The central Tactical level performs the computations and processing necessary to provide a symbolic-to-numeric interface between the two outer levels while implementing the behaviors capable of satisfying the goals assigned by the top level. Finally, the lower, Execution level contains the algorithms directly supporting the stability, safety, and hard real-time needs of the vehicle. These mechanisms and features are carried over from the design to the implementation through the use of specified programming paradigms, also explicit in the definition.

This chapter lists the contributions made to computer science by this research as well as suggestions for possible future extensions.

A. RESEARCH CONTRIBUTIONS

Control of autonomous vehicles encompasses a bewildering array of problem domains, from classical robotics and modern control theory to automated reasoning. Research has tended to focus on specific aspects of the overall control problem, such as navigation, path replanning, fault identification and isolation, sensor integration, and modeling at the expense of the global issue of system control. As autonomous systems have gained in sophistication due, in part, to breakthroughs in hardware technology, the lack of a general control software architecture to coordinate and organize the many diverse software subsystems has become evident.

From this need has emerged the field of research dedicated to intelligent control of robots, both stationary and mobile [Ref. 6]. The seminal work of Saridis [Ref. 37] argued for a multi-disciplinary, multi-level approach to software architectures; NASREM [Ref. 43] relied on a strict, top-down, hierarchical view; and Brooks [Ref. 31], partly as a philosophical challenge to the established robotics community, eschewed traditional approaches for a non-representational, behaviorally-layered architecture.

It has become apparent that these approaches, in their purest form, cannot provide the necessary flexibility of performance required and expected of autonomous vehicles [Ref. 55]. As a result, hybrid software architectures have begun to appear which exhibit both deliberative and reflexive components. Layered architectures have been fitted with top-level behavior-sequencing mechanisms [Ref. 56][Ref. 193], while hierarchical systems have integrated subsumptionist concepts at their lower levels [Ref. 45][Ref. 149].

Despite obvious progress, important problems still remain, as many suggested hybrid architectures have yet to be fully implemented. In addition, the ability to readily modify vehicle missions has been, for the most part, inadequately addressed. The Rational Behavior Model, a tri-level control software architecture, has been developed with the solution to these particular problems in mind.

As a way of bridging the gap from concept to implementation, RBM provides guidance to the autonomous vehicle system designer by explicitly specifying the programming paradigm and operational characteristics of each level in its definition. This was done to avoid the over-generalization of the solution, a common trait of existing architectures accounting for their less than complete instantiation. Many of these multi-level architectures, some recognized in the literature for a decade, remain only partially realized because of the lack of a perceived development strategy [Ref. 7][Ref. 39][Ref. 43]. Others have proponents whose research interests emphasize only one facet of the architecture [Ref. 37][Ref. 194]. In this dissertation, a total implementation of a multi-level control software architecture is presented.

Related to this issue is the importance RBM places on the interfaces between its three levels. While many architectures rely on the use of a blackboard data structure for the support of inter-level communications [Ref. 43][Ref. 61][Ref. 195], RBM requires simple, well-defined communications paths for the transmission of commands and data. The motivation behind this decision was three-fold: to contribute to the isolation of responsibility at each level; to minimize the affect an altered state or a software change at one level has upon another; and to provide for the potential of integrating RBM with an existing control system.

To facilitate the reconfiguration of the mission, it is necessary to separate mission logic from mission implementation. RBM accomplishes this by isolating this logic in the Strategic level. In other words, the portion of the architecture that specifies what the vehicle is to accomplish is completely divorced from the behavioral aspect of how the desired goals are to be attained. In addition, by excluding explicit state variables from the strategic level, the potential for undesired side-effects and undue complexity is avoided, further enhancing mission development and alteration. Several architectures have attempted similar solutions, but have either sacrificed simplicity for expressive power [Ref. 194] or have not provided a sufficient mechanism for reconfiguration [Ref. 56].

B. SUGGESTIONS FOR FUTURE RESEARCH

This research discussed the relationship between the two primary approaches to software architecture construction and the form of chaining used by their respective automated reasoners. From a pure logic standpoint, forward and backward chaining are merely two possible methods to solve a given problem. However, although the end results of each approach will be the same, the internal steps each takes in its path to the solution may differ significantly. In terms of the control of autonomous vehicles, these internal steps refer to the order in which behaviors and their associated side-effects are applied to the problem. This sequence of side-effects is often just as important to the designer and user of autonomous systems as the confidence in ultimate mission achievement. So, whereas a forward-

chaining architecture may attain the desired end goal assigned to it, the level of performance displayed during execution may be unacceptable.

The cause of this can be traced to the need for these systems to resolve conflicts associated with simultaneously activated behaviors. In many cases, conflicts are handled through the assignment of priorities, giving preference to the behavior considered most likely to move the state of the mission closer to the goal. Backward-chaining, goal-driven systems are characterized by their lack of conflicts with respect to behavior activations. This approach to reasoning, used by RBM, reflects directly the process of goal decomposition and the implied ordering of goals produced.

Despite the considerations outlined above, the majority of research in the field of autonomous vehicle control has been from the forward-chaining perspective. For this reason, RBM provides the option for a forward-chaining Strategic level. Should this option be exercised, however, the requirement remains that the primitive goals passed to the Tactical level be in the same sequence as would be achieved through backward chaining. Preliminary research into an algorithmic translation of a goal-driven solution to a forward-chaining implementation has resulted in the concept of the State Transition Diagram with Path Priorities [Ref. 140]. Development of such an algorithm is seen as a significant step towards merging the two approaches used in autonomous vehicles for the representation of human knowledge.

At the Tactical level, further exploration and possible refinement is warranted into the characteristics of the object hierarchy. Although ideally suited for a concurrent executing environment, the implementation detailed in this dissertation did not utilize the facility of multitasking provided by the Ada programming language. At issue here are performance enhancements and the potential impact concurrent programming has on software complexity and interaction. From the experimental results of Chapter VI, it was found that the performance of the simulation was virtually identical regardless of whether execution took place on two or three processors. Accounting for this is the minimal impact of inter-processor communications and the sequential relationship between the two levels. The conclusion

can then be drawn that for RBM implementations consisting of only static objects at the Tactical level, one processor is sufficient to host both the Strategic and Tactical levels. This would not be true for instantiations of RBM involving true parallelism at the Tactical level.

There exists a need to join this work to that done with pre-mission operator interfaces [Ref. 189]. These systems are designed to automatically generate the path(s) to be taken by the vehicle in the completion of its mission. These paths are typically in the form of lists of waypoints. This process is seen as an adjunct to the task of mission decomposition for the determination of primitive goals. In the researched reported in this dissertation, path planning and mission definition were handled in separate steps.

Another potential area of research relating to the expression of missions within the RBM framework is the development of language translators for the Strategic level. In the instantiations described in this work, Prolog and CLIPS were selected as examples of rule-based languages capable of effectively representing compiled human knowledge. Furthermore, these languages are made even more attractive by their inclusion of an inference mechanism responsible for the control of rule activation and firing. Once the mission has been defined to a sufficient degree, it is conceivable, and may even be desirable, to employ a language such as Lisp or Ada as the implementation language at the top level. For example, a complete, executable mission program written in Prolog may be available while circumstances dictate that some other language be used. Manual translations of this type indicate that this process may lend itself to automation [Ref. 196]. Of course, when generating non-rule-based language systems, the translator must bear the responsibility of representing knowledge in an effective way while properly emulating the relationship that exists between rule base and inference engine.

As presently defined, the Strategic level, once encoded, is immutable. This view derives from the paramount importance of predictability in vehicles designed for military applications. However, the applicability to RBM of machine learning resulting in modified rule sets better able to cope with uncertain environments may prove to be an important extension. In related research, an approach to the testing and evaluation of autonomous vehi-

cle controller performance using genetic algorithms has been recently demonstrated [Ref. 197]. These techniques are seen as a way to measure the robustness of an intelligent controller by subjecting it to a number of adaptively-chosen fault scenarios. An investigation into the integration of this research with the implementation of RBM is certainly warranted.

In the final analysis, the success of a software architecture is measured in terms of how well it provides structure and ease of interaction to the many software components while preventing undesirable interference or interaction between them. Since a feel for the "mathematical" correctness of a software architecture is not feasible, the system must be implemented on a real vehicle and its resulting behavior observed before full validation can be claimed. Unfortunately, the time schedules of the author and the rebuild schedule of the NPS AUV did not coincide sufficiently to allow this. An experimental validation on a detailed simulation of the vehicle is not without value, however, and much doctoral-level research into architectural approaches to autonomous vehicle control have relied on it [Ref. 60][Ref. 62][Ref. 198]. Nevertheless, the results demonstrated by the actual NPS AUV running the Florida search-and-rescue mission under the overall control of RBM will provide an important epilog to this research.

APPENDIX A. STRATEGIC LEVEL PROGRAM LISTINGS

1. PROLOG IMPLEMENTATION (RBM-B)

```
/* Strategic Level for the RBM AUV Mission Controller/Coordinator
 by Byrnes, Kwak, Healey, Marco for use in the Florida Mission

 Version: 2.5 Dec 15, 1992*/

/* ----------MISSION SPECIFICATION FOR SEARCH AND RESCUE---------- */

:- compile(library(not)).          /* A Quintus Prolog library providing "not" predicate */
:- compile(floridaforeign).        /* This file contains the foreign language interface */

initialize :- ready_vehicle_for_launch_p(ANS1), ANS1 == 1,
              select_first_waypoint(ANS2).
initialize :- alert_user(ANS), fail.

mission :- in_transit_p(ANS1), ANS1 == 1, transit, !, transit_done_p(ANS2), ANS2 == 1,
           fail.
mission :- in_search_p(ANS1), ANS1 == 1, search, !, search_done_p(ANS2),ANS2 == 1,
           fail.
mission :- in_task_p(ANS1), ANS1 == 1, task, !, task_done_p(ANS2), ANS2 == 1, fail.
mission :- in_return_p(ANS1), ANS1 == 1, return, !, return_done_p(ANS2), ANS2 == 1,
           wait_for_recovery(ANS3).

transit :- waypoint_control.
transit :- surface(ANS1), wait_for_recovery(ANS2).

search :- do_search_pattern(ANS), ANS == 1.
search :- surface(ANS1), wait_for_recovery(ANS2).
```

```
task :- homing(ANS1), ANS1 == 1, drop_package(ANS2), ANS2 == 1,
        get_gps_fix(ANS3), ANS3 == 1, get_next_waypoint(ANS4), ANS4 == 1.
task :- surface(ANS1), wait_for_recovery(ANS2).

return :- waypoint_control.
return :- surface(ANS1), wait_for_recovery(ANS2).


/* ---------------------------- NPS AUV DOCTRINE ----------*/


execute_auv_mission :- initialize, repeat, mission.

waypoint_control :- not(critical_system_prob), get_waypoint_status, plan,
                    send_setpoints_and_modes(ANS).

get_waypoint_status :- gps_check, reach_waypoint_p(ANS1), ANS1 == 1,
                        get_next_waypoint(ANS2).
get_waypoint_status.

gps_check :- gps_needed_p(ANS1), ANS1 == 1, get_gps_fix(ANS1).
gps_check.

plan :- reduced_capacity_system_prob, global_replan.
plan :- near_uncharted_obstacle, local_replan.
plan.

near_uncharted_obstacle :- unknown_obstacle_p(ANS1), ANS1 == 1,
                            log_new_obstacle(ANS2).

local_replan :- loiter(ANS1), start_local_replanner(ANS2).

global_replan :- loiter(ANS1), start_global_replanner(ANS2).

critical_system_prob :- power_gone_p(ANS), ANS == 1.
```

```
critical_system_prob :- computer_system_inop_p(ANS), ANS == 1.
critical_system_prob :- propulsion_system_p(ANS), ANS == 1.
critical_system_prob :- steering_system_inop_p(ANS), ANS == 1.

reduced_capacity_system_prob :- diving_system_p(ANS), ANS == 1.
reduced_capacity_system_prob :- bouyancy_system_p(ANS), ANS == 1.
reduced_capacity_system_prob :- thruster_system_p(ANS), ANS == 1.
reduced_capacity_system_prob :- leak_test_p(ANS), ANS == 1.
reduced_capacity_system_prob :- payload_prob_p(ANS), ANS == 1.
```

2. CLIPS IMPLEMENTATION (RBM-F)

```
;*****************************************************************************
;*
;*  Title          : Strategic Level for the NPS AUV II
;*  Name           : strlev4.0
;*  Version        : 4.0
;*  Author         : Thomas Scholz
;*  Date           : 22 February 1993
;*  Revised        : 23 February 1993 - First good run at 11:45am
;*  System         : Sun UNIX
;*  Compiler       : Clips 5.1
;*  Description    : This program is the strategic level of the NPS AUV II,
;*           top level of the Rational Behavioral Model design
;*  Remarks        : Don't make any changes to this program!
;*           It runs fine on the NPS AUV Simulator on IRIS!
;*
;*****************************************************************************

;*********************NPS AUV - RBM Mission Controller/Coordinator***********

;********************************** Templates ******************************

(deftemplate execute-auv-mission
        (field state
                (type SYMBOL)
                (allowed-symbols        initialize mission done inactive)
                (default inactive)))

(deftemplate or-initialize
        (field state
                (type SYMBOL)
                (allowed-symbols        or_initialize_1
                                        or_initialize_2
                                        failed done inactive)
```

```
        (default inactive)))

(deftemplate initialize
        (field state
                (type SYMBOL)
                (allowed-symbols        start
                                        select_first_waypoint
                                        alert_user
                                        failed done inactive)
                (default inactive)))

(deftemplate or-mission
        (field state
                (type SYMBOL)
                (allowed-symbols        or_mission_1
                                        or_mission_2
                                        or_mission_3
                                        or_mission_4
                                        done inactive)
                (default inactive)))

(deftemplate mission
        (field state
                (type SYMBOL)
                (allowed-symbols        start
                                        transit in_transit_p transit_done_p
                                        search in_search_p search_done_p
                                        task in_task_p task_done_p
                                        return in_return_p return_done_p
                                        done inactive)
                (default inactive)))

(deftemplate transit
        (field state
                (type SYMBOL)
                (allowed-symbols        start
                                        waypoint_control
                                        surface
                                        done inactive)
```

```
        (default  inactive)))


(deftemplate or-transit
        (field state
                (type SYMBOL)
                (allowed-symbols        or_transit_1
                                        or_transit_2
                                        done  inactive)
                (default inactive)))


(deftemplate search
        (field state
                (type SYMBOL)
                (allowed-symbols        start
                                        do_search_pattern
                                        surface
                                        done  inactive)
                (default inactive)))


(deftemplate or-search
        (field state
                (type SYMBOL)
                (allowed-symbols        or_search_1
                                        or_search_2
                                        done  inactive)
                (default inactive)))


(deftemplate task
        (field state
                (type SYMBOL)
                (allowed-symbols        start
                                        homing  drop_package  get_gps_fix
                                        get_next_waypoint  surface
                                        done  inactive)
                (default inactive)))


(deftemplate or-task
        (field state
```

```
            (type SYMBOL)
            (allowed-symbols        or_task_1
                                    or_task_2
                                    done inactive)
            (default inactive)))

(deftemplate return
      (field state
            (type SYMBOL)
            (allowed-symbols        start
                                    waypoint_control
                                    surface
                                    done inactive)
            (default inactive)))

(deftemplate or-return
      (field state
            (type SYMBOL)
            (allowed-symbols        or_return_1
                                    or_return_2
                                    done inactive)
            (default inactive)))

(deftemplate waypoint-control
      (field state
            (type SYMBOL)
            (allowed-symbols        start
                                    crit_system_prob
                                    get_waypoint_status
                                    plan send_setpoints_and_modes
                                    done inactive)
            (default inactive)))

(deftemplate get-waypoint-status
      (field state
            (type SYMBOL)
            (allowed-symbols        start
                                    gps_check
                                    reach_waypoint
```

```
                                        get_next_waypoint
                                        done inactive)
                (default inactive)))


(deftemplate or-get-waypoint-status
        (field state
                (type SYMBOL)
                (allowed-symbols        or_get_waypoint_status_1
                                        or_get_waypoint_status_2
                                        done inactive)
                (default inactive)))


(deftemplate gps-check
        (field state
                (type SYMBOL)
                (allowed-symbols        start
                                        gps_needed
                                        get_gps_fix
                                        done inactive)
                (default inactive)))


(deftemplate or-gps-check
        (field state
                (type SYMBOL)
                (allowed-symbols        or_gps_check_1
                                        or_gps_check_2
                                        done inactive)
                (default inactive)))


(deftemplate plan
        (field state
                (type SYMBOL)
                (allowed-symbols        start
                                        red_cap_system_prob
                                        near_uncharted_obstacle
                                        global_replan
                                        local_replan
                                        done inactive)
                (default inactive)))
```

```
(deftemplate or-plan
        (field state
                (type SYMBOL)
                (allowed-symbols        or_plan_1
                                        or_plan_2
                                        or_plan_3
                                        done  inactive)
                (default inactive)))

(deftemplate near-uncharted-obstacle
        (field state
                (type SYMBOL)
                (allowed-symbols        start
                                        unknown_obstacle_p
                                        log_new_obstacle
                                        done  inactive)
                (default inactive)))

(deftemplate global-replan
        (field state
                (type SYMBOL)
                (allowed-symbols        start
                                        loiter
                                        start_global_replanner
                                        done  inactive)
                (default inactive)))

(deftemplate local-replan
        (field state
                (type SYMBOL)
                (allowed-symbols        start
                                        loiter
                                        start_local_replanner
                                        done  inactive)
                (default inactive)))

(deftemplate crit-system-prob
```

```
        (field state
                (type SYMBOL)
                (allowed-symbols        start
                                        power_gone_p
                                        computer_system_inop_p
                                        propulsion_system_p
                                        steering_system_inop_p
                                        done inactive)
                (default inactive)))

(deftemplate or-crit-system-prob
        (field state
                (type SYMBOL)
                (allowed-symbols        or_crit_system_prob_1
                                        or_crit_system_prob_2
                                        or_crit_system_prob_3
                                        or_crit_system_prob_4
                                        done inactive)
                (default inactive)))

(deftemplate red-cap-system-prob
        (field state
                (type SYMBOL)
                (allowed-symbols        start
                                        diving_system_p
                                        bouyancy_system_p
                                        thruster_system_p
                                        leak_test_p
                                        payload_prob_p
                                        done inactive)
                (default inactive)))

(deftemplate or-red-cap-system-prob
        (field state
                (type SYMBOL)
                (allowed-symbols        or_red_cap_system_prob_1
                                        or_red_cap_system_prob_2
                                        or_red_cap_system_prob_3
                                        or_red_cap_system_prob_4
                                        or_red_cap_system_prob_5
```

```
                                        done  inactive)
            (default inactive)))

;************************************ Rules ******************************

(defrule execute-auv-mission-10-s
        ?x <- (start)
=>
        (retract ?x)
        (assert (execute-auv-mission (state initialize))))


(defrule execute-auv-mission-10-e
        ?x <- (execute-auv-mission (state done))
=>
        (retract ?x)
        (reset)
        (assert (execute-auv-mission (state mission))))


(defrule execute-auv-mission-11-s
        (execute-auv-mission (state initialize))
=>
        (assert (initialize (state start))))


(defrule execute-auv-mission-11-e
        ?x <- (execute-auv-mission (state initialize))
        ?y <- (initialize (state done))
=>
        (retract ?x)
        (retract ?y)
        (assert (execute-auv-mission (state mission))))


(defrule execute-auv-mission-12-s
        (execute-auv-mission (state mission))
=>
        (assert (mission (state start))))
```

```
(defrule execute-auv-mission-12-e
        ?x <- (execute-auv-mission (state mission))
        ?y <- (mission (state done))
=>
        (retract ?x)
        (retract ?y)
        (assert (execute-auv-mission (state done))))


(defrule initialize-10-s
        (declare (salience 10))
        ?x <- (initialize (state start))
=>
        (retract ?x)
        (assert (or-initialize (state or_initialize_1)))
        (assert (initialize (state select_first_waypoint))))


(defrule initialize-10-e
        (declare (salience -10000))
        ?x <- (or-initialize (state or_initialize_1))
=>
        (retract ?x)
        (assert (or-initialize (state or_initialize_2))))


(defrule initialize-11
        (or-initialize (state or_initialize_1))
        (initialize (state select_first_waypoint))
        (test (= (ready_vehicle_for_launch) 1))
=>
        (select_first_waypoint)
        (assert (initialize (state done)))
        (printout t "* * * auv execution initialized * * *" crlf crlf))


(defrule initialize-20-s
        (or-initialize (state or_initialize_2))
=>
        (assert (initialize (state alert_user))))


(defrule initialize-20-e
```

```
        ?x <- (initialize (state alert_user))
=>
        (retract ?x)
        (alert_user)
        (printout t "*** initialize failed ***" crlf crlf)
        (assert (initialize (state failed))))


(defrule mission-10-s
        (declare (salience 30))
        ?x <- (mission (state start))
=>
        (retract ?x)
        (assert (or-mission (state or_mission_1)))
        (assert (mission (state transit))))


(defrule mission-10-e
        (declare (salience -1000))
        ?x <- (or-mission (state or_mission_1))
=>
        (retract ?x)
        (assert (or-mission (state or_mission_2))))


(defrule mission-11-s
        (or-mission (state or_mission_1))
        (mission (state transit))
        (test (= (in_transit_p) 1))
=>
        (assert (transit (state start))))


(defrule mission-11-e
        (or-mission (state or_mission_1))
        ?x <- (mission (state transit))
        ?y <- (transit (state done))
=>
        (retract ?x)
        (retract ?y)
        (assert (mission (state transit_done_p))))
```

```
(defrule mission-12
        (or-mission (state or_mission_1))
        ?x <- (mission (state transit_done_p))
        (test (= (transit_done_p) 1))
=>
        )


(defrule mission-20-s
        (declare (salience 20))
        (or-mission (state or_mission_2))
=>
        (assert (mission (state search))))


(defrule mission-20-e
        (declare (salience -1000))
        ?x <- (or-mission (state or_mission_2))
=>
        (retract ?x)
        (assert (or-mission (state or_mission_3))))


(defrule mission-21-s
        (or-mission (state or_mission_2))
        (mission (state search))
        (test (= (in_search_p) 1))
=>
        (assert (search (state start))))


(defrule mission-21-e
        (or-mission (state or_mission_2))
        ?x <- (mission (state search))
        ?y <- (search (state done))
=>
        (retract ?x)
        (retract ?y)
        (assert (mission (state search_done_p))))


(defrule mission-22
        (or-mission (state or_mission_2))
```

```
        ?x <- (mission (state search_done_p))
        (test (= (search_done_p) 1))
=>
        )


(defrule mission-30-s
        (declare (salience 10))
        (or-mission (state or_mission_3))
=>
        (assert (mission (state task))))


(defrule mission-30-e
        (declare (salience -1000))
        ?x <- (or-mission (state or_mission_3))
=>
        (retract ?x)
        (assert (or-mission (state or_mission_4))))


(defrule mission-31-s
        (or-mission (state or_mission_3))
        (mission (state task))
        (test (= (in_task_p) 1))
=>
        (assert (task (state start))))


(defrule mission-31-e
        (or-mission (state or_mission_3))
        ?x <- (mission (state task))
        ?y <- (task (state done))
=>
        (retract ?x)
        (retract ?y)
        (assert (mission (state task_done_p))))


(defrule mission-32
        (or-mission (state or_mission_3))
        ?x <- (mission (state task_done_p))
        (test (= (task_done_p) 1))
```

```
=>
        )


(defrule mission-40-s
        (declare (salience 0))
        (or-mission (state or_mission_4))
=>
        (assert (mission (state return))))


(defrule mission-40-e
        (declare (salience -1000))
        ?x <- (or-mission (state or_mission_4))
=>
        (retract ?x)
        (assert (mission (state done))))


(defrule mission-41-s
        (or-mission (state or_mission_4))
        (mission (state return))
        (test (= (in_return_p) 1))
=>
        (assert (return (state start))))


(defrule mission-41-e
        (or-mission (state or_mission_4))
        ?x <- (mission (state return))
        ?y <- (return (state done))
=>
        (retract ?x)
        (retract ?y)
        (assert (mission (state return_done_p))))


(defrule mission42
        (or-mission (state or_mission_4))
        ?x <- (mission (state return_done_p))
        (test (= (return_done_p) 1))
=>
        (wait_for_recovery)
```

```
        (retract ?x))


(defrule transit-10-s
        (declare (salience 10))
        ?x <- (transit (state start))
=>
        (retract ?x)
        (assert (or-transit (state or_transit_1))))


(defrule transit-10-e1
     (declare (salience 9))
     (transit (state done))
     ?x <- (or-transit (state or_transit_1))
=>
     (retract ?x))


(defrule transit-10-e
        (declare (salience -100))
        ?x <- (or-transit (state or_transit_1))
=>
        (retract ?x)
        (assert (or-transit (state or_transit_2))))


(defrule transit-11-s
        (or-transit (state or_transit_1))
=>
        (assert (transit (state waypoint_control)))
        (assert (waypoint-control (state start))))


(defrule transit-11-end
        (or-transit (state or_transit_1))
        ?y <- (transit (state waypoint_control))
        ?z <- (waypoint-control (state done))
=>
        (retract ?y)
        (retract ?z)
        (assert (transit (state done))))
```

```
(defrule transit-21-s
        (or-transit (state or_transit_2))
=>
        (assert (transit (state surface))))


(defrule transit-21-end
        ?x <- (or-transit (state or_transit_2))
        ?y <- (transit (state surface))
=>
        (surface)
        (retract ?x)
        (retract ?y)
        (assert (transit (state done))))


(defrule search-10-s
        (declare (salience 10))
        ?x <- (search (state start))
=>
        (retract ?x)
        (assert (or-search (state or_search_1))))


(defrule search-10-e1
     (declare (salience 9))
        (search (state done))
     ?x <- (or-search (state or_search_1))
=>
     (retract ?x))


(defrule search-10-e
        (declare (salience -100))
        ?x <- (or-search (state or_search_1))
=>
        (retract ?x)
        (assert (or-search (state or_search_2))))


(defrule search-11-s
        (or-search (state or_search_1))
```

```
=>
        (assert (search (state do_search_pattern))))


(defrule search-11-end
        (or-search (state or_search_1))
        ?y <- (search (state do_search_pattern))
        (test (= (do_search_pattern) 1))
=>
        (retract ?y)
        (assert (search (state done))))


(defrule search-21-s
        (or-search (state or_search_2))
=>
        (assert (search (state surface))))


(defrule search-21-end
        ?x <- (or-search (state or_search_2))
        ?y <- (search (state surface))
=>
        (surface)
        (retract ?x)
        (retract ?y)
        (assert (search (state done))))


(defrule task-10-s
        (declare (salience 10))
        ?x <- (task (state start))
=>
        (retract ?x)
        (assert (or-task (state or_task_1))))


(defrule task-10-e1
     (declare (salience 9))
        (task (state done))
     ?x <- (or-task (state or_task_1))
=>
     (retract ?x))
```

```
(defrule task-10-e
        (declare (salience -100))
        ?x <- (or-task (state or_task_1))
=>
        (retract ?x)
        (assert (or-task (state or_task_2))))


(defrule task-11-s
        (or-task (state or_task_1))
=>
        (assert (task (state homing))))


(defrule task-11-e
        (or-task (state or_task_1))
        ?x <- (task (state homing))
        (test (= (homing) 1))
=>
        (retract ?x)
        (assert (task (state drop_package))))


(defrule task-12
        (or-task (state or_task_1))
        ?x <- (task (state drop_package))
        (test (= (drop_package) 1))
=>
        (retract ?x)
        (assert (task (state get_gps_fix))))


(defrule task-13
        (or-task (state or_task_1))
        ?x <- (task (state get_gps_fix))
        (test (= (get_gps_fix) 1))
=>
        (retract ?x)
        (assert (task (state get_next_waypoint))))
```

```
(defrule task-14-end
        (or-task (state or_task_1))
        ?y <- (task (state get_next_waypoint))
        (test (= (get_next_waypoint) 1))
=>
        (retract ?y)
        (assert (task (state done))))


(defrule task-21-s
        (or-task (state or_task_2))
=>
        (assert (task (state surface))))


(defrule task-21-end
        ?x <- (or-task (state or_task_2))
        ?y <- (task (state surface))
=>
        (surface)
        (retract ?x)
        (retract ?y)
        (assert (task (state done)))
        (printout t "*** starting task failed ***" crlf)
        (printout t crlf "+++ auv is surfacing [due to problems] +++" crlf)
        (printout t "+++ [and] mission execution terminated +++" crlf crlf))


(defrule return-10-s
        (declare (salience 10))
        ?x <- (return (state start))
=>
        (retract ?x)
        (assert (or-return (state or_return_1))))


(defrule return-10-e1
        (declare (salience 9))
        (return (state done))
        ?x <- (or-return (state or_return_1))
=>
        (retract ?x))
```

```
(defrule return-10-e
        (declare (salience -100))
        ?x <- (or-return (state or_return_1))
=>
        (retract ?x)
        (assert (or-return (state or_return_2))))


(defrule return-11-s
        (or-return (state or_return_1))
=>
        (assert (return (state waypoint_control)))
        (assert (waypoint-control (state start))))


(defrule return-11-end
        (or-return (state or_return_1))
        ?y <- (return (state waypoint_control))
        ?z <- (waypoint-control (state done))
=>
        (retract ?y)
        (retract ?z)
        (assert (return (state done))))


(defrule return-21-s
        (or-return (state or_return_2))
=>
        (assert (return (state surface))))


(defrule return-21-end
        ?x <- (or-return (state or_return_2))
        ?y <- (return (state surface))
=>
        (surface)
        (retract ?x)
        (retract ?y)
        (assert (return (state done))))


(defrule waypoint-control-10-s
```

```
        ?x <- (waypoint-control (state start))
=>
        (retract ?x)
        (assert (waypoint-control (state crit_system_prob))))


(defrule waypoint-control-11-s
        (waypoint-control (state crit_system_prob))
=>
        (assert (crit-system-prob (state start))))


(defrule waypoint-control-11-e
        ?x <- (waypoint-control (state crit_system_prob))
        (not (crit-system-prob (state done)))
=>
        (retract ?x)
        (assert (waypoint-control (state get_waypoint_status))))


(defrule waypoint-control-12-s
        (waypoint-control (state get_waypoint_status))
=>
        (assert (get-waypoint-status (state start))))


(defrule waypoint-control-12-e
        ?x <- (waypoint-control (state get_waypoint_status))
        ?y <- (get-waypoint-status (state done))
=>
        (retract ?x)
        (retract ?y)
        (assert (waypoint-control (state plan))))


(defrule waypoint-control-13-s
        (waypoint-control (state plan))
=>
        (assert (plan (state start))))


(defrule waypoint-control-13-e
        ?x <- (waypoint-control (state plan))
```

```
        ?y <- (plan (state done))
=>
        (retract ?x)
        (retract ?y)
        (assert (waypoint-control (state send_setpoints_and_modes))))


(defrule waypoint-control-14-end
        ?x <- (waypoint-control (state send_setpoints_and_modes))
=>
        (send_setpoints_and_modes)
        (retract ?x)
        (assert (waypoint-control (state done))))


(defrule get-waypoint-status-10-s
        (declare (salience 10))
        ?x <- (get-waypoint-status (state start))
=>
        (retract ?x)
        (assert (or-get-waypoint-status (state or_get_waypoint_status_1))))


(defrule get-waypoint-status-10-e1
        (declare (salience -9))
        (get-waypoint-status (state done))
        ?x <- (or-get-waypoint-status (state or_get_waypoint_status_1))
=>
        (retract ?x))


(defrule get-waypoint-status-10-e
        (declare (salience -10))
        ?x <- (or-get-waypoint-status (state or_get_waypoint_status_1))
=>
        (retract ?x)
        (assert (or-get-waypoint-status (state or_get_waypoint_status_2))))


(defrule get-waypoint-status-11-s
        (or-get-waypoint-status (state or_get_waypoint_status_1))
=>
        (assert (get-waypoint-status (state gps_check)))
```

```
        (assert (gps-check (state start))))


(defrule get-waypoint-status-11-e
        (or-get-waypoint-status (state or_get_waypoint_status_1))
        ?x <- (get-waypoint-status (state gps_check))
        ?y <- (gps-check (state done))
=>
        (retract ?x)
        (retract ?y)
        (assert (get-waypoint-status (state reach_waypoint))))


(defrule get-waypoint-status-12-end
        (or-get-waypoint-status (state or_get_waypoint_status_1))
        ?y <- (get-waypoint-status (state reach_waypoint))
        (test (= (reach_waypoint) 1))
=>
        (get_next_waypoint)
        (retract ?y)
        (assert (get-waypoint-status (state done))))


(defrule get-waypoint-status-21-end
        ?x <- (or-get-waypoint-status (state or_get_waypoint_status_2))
=>
        (retract ?x)
        (assert (get-waypoint-status (state done))))


(defrule gps-check-10-s
        (declare (salience 10))
        ?x <- (gps-check (state start))
=>
        (retract ?x)
        (assert (or-gps-check (state or_gps_check_1))))


(defrule gps-check-10-e1
        (declare (salience 9))
        (gps-check (state done))
        ?x <- (or-gps-check (state or_gps_check_1))
=>
```

```
        (retract ?x))


(defrule gps-check-10-e
        (declare (salience -10))
        ?x <- (or-gps-check (state or_gps_check_1))
=>
        (retract ?x)
        (assert (or-gps-check (state or_gps_check_2))))


(defrule gps-check-11-s
        (or-gps-check (state or_gps_check_1))
=>
        (assert (gps-check (state gps_needed))))


(defrule gps-check-11-end
        (or-gps-check (state or_gps_check_1))
        ?y <- (gps-check (state gps_needed))
        (test (= (gps_needed) 1))
=>
        (get_gps_fix)
        (retract ?y)
        (assert (gps-check (state done))))


(defrule gps-check-21-end
        ?x <- (or-gps-check (state or_gps_check_2))
=>
        (retract ?x)
        (assert (gps-check (state done))))


(defrule plan-10-s
        (declare (salience 10))
        ?x <- (plan (state start))
=>
        (retract ?x)
        (assert (or-plan (state or_plan_1))))


(defrule plan-20-s
```

```
        (declare (salience -10))
        ?x <- (or-plan (state or_plan_1))
=>
        (retract ?x)
        (assert (or-plan (state or_plan_2))))


(defrule plan-30-s
        (declare (salience -20))
        ?x <- (or-plan (state or_plan_2))
=>
        (retract ?x)
        (assert (or-plan (state or_plan_3))))


(defrule plan-11-s
        (or-plan (state or_plan_1))
=>
        (assert (plan (state red_cap_system_prob)))
        (assert (red-cap-system-prob (state start))))


(defrule plan-11-e
        (or-plan (state or_plan_1))
        ?x <- (plan (state red_cap_system_prob))
        ?y <- (red-cap-system-prob (state done))
=>
        (retract ?x)
        (retract ?y)
        (assert (plan (state global_replan)))
        (assert (global-replan (state start))))


(defrule plan-12-s
        (or-plan (state or_plan_1))
        ?x <- (plan (state global_replan))
        ?y <- (global-replan (state done))
=>
        (retract ?x)
        (retract ?y)
        (assert (plan (state done))))
```

```
(defrule plan-21-s
        (or-plan (state or_plan_2))
=>
        (assert (plan (state near_uncharted_obstacle)))
        (assert (near-uncharted-obstacle (state start))))


(defrule plan-21-e
        (or-plan (state or_plan_2))
        ?x <- (plan (state near_uncharted_obstacle))
        ?y <- (near-uncharted-obstacle (state done))
=>
        (retract ?x)
        (retract ?y)
        (assert (plan (state local_replan)))
        (assert (local-replan (state start))))


(defrule plan-22-end
        ?x <- (or-plan (state or_plan_2))
        ?y <- (plan (state local_replan))
        ?z <- (local-replan (state done))
=>
        (retract ?x)
        (retract ?y)
        (retract ?z)
        (assert (plan (state done))))


(defrule plan-31-end
        ?x <- (or-plan (state or_plan_3))
=>
        (retract ?x)
        (assert (plan (state done))))


(defrule near-uncharted-obstacle-10-s
        ?x <- (near-uncharted-obstacle (state start))
=>
        (retract ?x)
        (assert (near-uncharted-obstacle (state unknown_obstacle_p))))
```

```
(defrule near-uncharted-obstacle-11-s
        ?x <- (near-uncharted-obstacle (state unknown_obstacle_p))
        (test (= (unknown_obstacle_p) 1))
=>
        (retract ?x)
        (assert (near-uncharted-obstacle (state log_new_obstacle))))


(defrule near-uncharted-obstacle-12-end
        ?x <- (near-uncharted-obstacle (state log_new_obstacle))
=>
        (log_new_obstacle)
        (assert (near-uncharted-obstacle (state done))))


(defrule local-replan-10-s
        ?x <- (local-replan (state start))
=>
        (retract ?x)
        (assert (local-replan (state loiter))))


(defrule local-replan-11-s
        ?x <- (local-replan (state loiter))
=>
        (loiter)
        (retract ?x)
        (assert (local-replan (state start_local_replanner))))


(defrule local-replan-12-end
        ?x <- (local-replan (state start_local_replanner))
=>
        (start_local_replanner)
        (retract ?x)
        (assert (local-replan (state done))))


(defrule global-replan-10-s
        ?x <- (global-replan (state start))
=>
        (retract ?x)
        (assert (global-replan (state loiter))))
```

```
(defrule global-replan-11-s
        ?x <- (global-replan (state loiter))
=>
        (loiter)
        (retract ?x)
        (assert (global-replan (state start_global_replanner))))


(defrule global-replan-12-end
        ?x <- (global-replan (state start_global_replanner))
=>
        (start_global_replanner)
        (retract ?x)
        (assert (global-replan (state done))))


(defrule crit-system-prob-10-s
        (declare (salience 10))
        ?x <- (crit-system-prob (state start))
=>
        (retract ?x)
        (assert (or-crit-system-prob (state or_crit_system_prob_1))))


(defrule crit-system-prob-10-end
        (declare (salience 1))
        ?x <- (or-crit-system-prob (state or_crit_system_prob_4))
        ?y <- (crit-system-prob (state steering_system_inop_p))
=>
        (retract ?x)
        (retract ?y))


(defrule crit-system-prob-20-s
        (declare (salience 9))
        ?x <- (or-crit-system-prob (state or_crit_system_prob_1))
        ?y <- (crit-system-prob (state power_gone_p))
=>
        (retract ?x)
        (retract ?y)
        (assert (or-crit-system-prob (state or_crit_system_prob_2))))
```

```
(defrule crit-system-prob-30-s
        (declare (salience 8))
        ?x <- (or-crit-system-prob (state or_crit_system_prob_2))
        ?y <- (crit-system-prob (state computer_system_inop_p))
=>
        (retract ?x)
        (retract ?y)
        (assert (or-crit-system-prob (state or_crit_system_prob_3))))


(defrule crit-system-prob-40-s
        (declare (salience 7))
        ?x <- (or-crit-system-prob (state or_crit_system_prob_3))
        ?y <- (crit-system-prob (state propulsion_system_p))
=>
        (retract ?x)
        (retract ?y)
        (assert (or-crit-system-prob (state or_crit_system_prob_4))))


(defrule crit-system-prob-11-s
        (or-crit-system-prob (state or_crit_system_prob_1))
=>
        (assert (crit-system-prob (state power_gone_p))))


(defrule crit-system-prob-11-end
        (declare (salience 10))
        ?x <- (or-crit-system-prob (state or_crit_system_prob_1))
        ?y <- (crit-system-prob (state power_gone_p))
        (test (= (power_gone_p) 1))
=>
        (retract ?x)
        (retract ?y)
        (assert (crit-system-prob (state done))))


(defrule crit-system-prob-21-s
        (or-crit-system-prob (state or_crit_system_prob_2))
=>
        (assert (crit-system-prob (state computer_system_inop_p))))
```

```
(defrule crit-system-prob-21-end
        (declare (salience 9))
        ?x <- (or-crit-system-prob (state or_crit_system_prob_2))
        ?y <- (crit-system-prob (state computer_system_inop_p))
        (test (= (computer_system_inop_p) 1))
=>
        (retract ?x)
        (retract ?y)
        (assert (crit-system-prob (state done))))


(defrule crit-system-prob-31-s
        (or-crit-system-prob (state or_crit_system_prob_3))
=>
        (assert (crit-system-prob (state propulsion_system_p))))


(defrule crit-system-prob-31-end
        (declare (salience 8))
        ?x <- (or-crit-system-prob (state or_crit_system_prob_3))
        ?y <- (crit-system-prob (state propulsion_system_p))
        (test (= (propulsion_system_p) 1))
=>
        (retract ?x)
        (retract ?y)
        (assert (crit-system-prob (state done))))


(defrule crit-system-prob-41-s
        (or-crit-system-prob (state or_crit_system_prob_4))
=>
        (assert (crit-system-prob (state steering_system_inop_p))))


(defrule crit-system-prob-41-end
        (declare (salience 7))
        ?x <- (or-crit-system-prob (state or_crit_system_prob_4))
        ?y <- (crit-system-prob (state steering_system_inop_p))
        (test (= (steering_system_inop_p) 1))
=>
        (retract ?x)
```

```
        (retract ?y)
        (assert (crit-system-prob (state done))))


(defrule red-cap-system-prob-10-s
        (declare (salience 10))
        ?x <- (red-cap-system-prob (state start))
=>
        (retract ?x)
        (assert (or-red-cap-system-prob (state or_red_cap_system_prob_1))))


(defrule red-cap-system-prob-10-end
        ?x <- (or-red-cap-system-prob (state or_red_cap_system_prob_5))
        ?y <- (red-cap-system-prob (state payload_prob_p))
=>
        (retract ?x)
        (retract ?y)
        (assert (red-cap-system-prob (state done)))


(defrule red-cap-system-prob-20-s
        (declare (salience 9))
        ?x <- (or-red-cap-system-prob (state or_red_cap_system_prob_1))
        ?y <- (red-cap-system-prob (state diving_system_p))
=>
        (retract ?x)
        (retract ?y)
        (assert (or-red-cap-system-prob (state or_red_cap_system_prob_2))))


(defrule red-cap-system-prob-30-s
        (declare (salience 8))
        ?x <- (or-red-cap-system-prob (state or_red_cap_system_prob_2))
        ?y <- (red-cap-system-prob (state bouyancy_system_p))
=>
        (retract ?x)
        (retract ?y)
        (assert (or-red-cap-system-prob (state or_red_cap_system_prob_3))))


(defrule red-cap-system-prob-40-s
        (declare (salience 7))
```

```
        ?x <- (or-red-cap-system-prob (state or_red_cap_system_prob_3))
        ?y <- (red-cap-system-prob (state thruster_system_p))
=>
        (retract ?x)
        (retract ?y)
        (assert (or-red-cap-system-prob (state or_red_cap_system_prob_4))))


(defrule red-cap-system-prob-50-s
        (declare (salience 6 ))
        ?x <- (or-red-cap-system-prob (state or_red_cap_system_prob_4))
        ?y <- (red-cap-system-prob (state leak_test_p))
=>
        (retract ?x)
        (retract ?y)
        (assert (or-red-cap-system-prob (state or_red_cap_system_prob_5))))


(defrule red-cap-system-prob-11-s
        (or-red-cap-system-prob (state or_red_cap_system_prob_1))
=>
        (assert (red-cap-system-prob (state diving_system_p))))


(defrule red-cap-system-prob-11-end
        (declare (salience 10))
        ?x <- (or-red-cap-system-prob (state or_red_cap_system_prob_1))
        ?y <- (red-cap-system-prob (state diving_system_p))
        (test (= (diving_system_p) 1))
=>
        (retract ?x)
        (retract ?y)
        (assert (red-cap-system-prob (state done))))


(defrule red-cap-system-prob-21-s
        (or-red-cap-system-prob (state or_red_cap_system_prob_2))
=>
        (assert (red-cap-system-prob (state bouyancy_system_p))))


(defrule red-cap-system-prob-21-end
        (declare (salience 9))
```

```
        ?x <- (or-red-cap-system-prob (state or_red_cap_system_prob_2))
        ?y <- (red-cap-system-prob (state bouyancy_system_p))
        (test (= (bouyancy_system_p) 1))
=>
        (retract ?x)
        (retract ?y)
        (assert (red-cap-system-prob (state done))))


(defrule red-cap-system-prob-31-s
        (or-red-cap-system-prob (state or_red_cap_system_prob_3))
=>
        (assert (red-cap-system-prob (state thruster_system_p))))


(defrule red-cap-system-prob-31-end
        (declare (salience 8))
        ?x <- (or-red-cap-system-prob (state or_red_cap_system_prob_3))
        ?y <- (red-cap-system-prob (state thruster_system_p))
        (test (= (thruster_system_p) 1))
=>
        (retract ?x)
        (retract ?y)
        (assert (red-cap-system-prob (state done))))


(defrule red-cap-system-prob-41-s
        (or-red-cap-system-prob (state or_red_cap_system_prob_4))
=>
        (assert (red-cap-system-prob (state leak_test_p))))


(defrule red-cap-system-prob-41-end
        (declare (salience 7))
        ?x <- (or-red-cap-system-prob (state or_red_cap_system_prob_4))
        ?y <- (red-cap-system-prob (state leak_test_p))
        (test (= (leak_test_p) 1))
=>
        (retract ?x)
        (retract ?y)
        (assert (red-cap-system-prob (state done))))
```

```
(defrule red-cap-system-prob-51-s
        (or-red-cap-system-prob (state or_red_cap_system_prob_5))
=>
        (assert (red-cap-system-prob (state payload_prob_p))))


(defrule red-cap-system-prob-51-end
        (declare (salience 6))
        ?x <- (or-red-cap-system-prob (state or_red_cap_system_prob_5))
        ?y <- (red-cap-system-prob (state payload_prob_p))
        (test (= (payload_prob_p) 1))
=>
        (retract ?x)
        (retract ?y)
        (assert (red-cap-system-prob (state done))))


;*****************************************************************************
```

3. TRACES OF THE EXECUTION OF THE SEARCH AND RESCUE MISSION

The first trace represents the chain of inference derived from the logic embodied in the rule set of the Prolog and CLIPS programs for a complete, four-phase search_-and_rescue mission in which no environmental or systemic abnormalities were encountered.

To start the mission using the backward-chaining Prolog implementation, a query "execute_auv_mission" is presented to the Prolog system. The inference engine searches the rule set for a rule head that matches this query and attempts to satisfy this rule's subgoals in sequence from left to right:

To start the mission using the forward-chaining CLIPS implementation, the (start) fact is first asserted to the fact-list, followed by the command (run). The CLIPS inference engine matches the current facts with the left-hand sides of the rules; responses to primitive goals are evaluated as part of the attempt to satisfy the conditional part of the rule.

The traces listed herein are identical for both the Prolog and CLIPS implementations.

Entered the function ready_vehicle_for_launch_p

Entered the function select_first_waypoint

Following initialization, the "transit" phase is entered:

Entered the function in_transit_p

Entered the function power_gone_p

Entered the function computer_system_inop_p

Entered the function propulsion_system_p

Entered the function steering_system_inop_p

Entered the function gps_needed_p

Entered the function reach_waypoint_p

Entered the function diving_system_p

Entered the function bouyancy_system_p

Entered the function thruster_system_p

Entered the function leak_test_p

Entered the function payload_prob_p

Entered the function unknown_obstacle_p

Entered the function send_setpoints_and_modes

Entered the function transit_done_p

Entered the function in_transit_p

Entered the function power_gone_p

Entered the function computer_system_inop_p

.
.
.

and so on, occasionally reaching the primitive goal "get_next_waypoint" when the query goal "reach_waypoint" receives a TRUE response. When the transit phase is complete, i.e., when the vehicle arrives at the search area, the search phase begins:

Entered the function in_transit_p

Entered the function in_search_p

Entered the function do_search_pattern

Entered the function search_done_p

When the target has been located and identified, the search phase ends and the task phase commences:

Entered the function in_transit_p

Entered the function in_search_p

Entered the function in_task_p

Entered the function homing

Entered the function drop_package

Entered the function get_gps_fix

Entered the function get_next_waypoint

Entered the function task_done_p

Finally, following the completipon of the task and subsequent gps location fix, the next waypoint is selected, corresponding to the first waypoint on the return path. This prepares the vehicle for the return phase of the mission:

Entered the function in_transit_p

Entered the function in_search_p

Entered the function in_task_p

Entered the function in_return_p

Entered the function power_gone_p

Entered the function computer_system_inop_p

Entered the function propulsion_system_p

Entered the function steering_system_inop_p

Entered the function gps_needed_p

Entered the function reach_waypoint_p

Entered the function diving_system_p

Entered the function bouyancy_system_p

Entered the function thruster_system_p

Entered the function leak_test_p

Entered the function payload_prob_p

Entered the function unknown_obstacle_p

Entered the function send_setpoints_and_modes

Entered the function return_done_p

Entered the function in_transit_p

Entered the function in_search_p

Entered the function in_task_p

Entered the function in_return_p

.
.
.

The logic used is precisely the same as the outbound transit, because the return is nothing more than a second transit phase. Therefore, this loop will also visit the primitive goal "get_next_waypoint" if multiple waypoints have been specified for the return leg of the journey. Upon reaching the final goal, the return phase is marked complete and the vehicle prepares itself for retrieval:

Entered the function in_transit_p

Entered the function in_search_p

Entered the function in_task_p

Entered the function in_return_p

Entered the function power_gone_p

Entered the function computer_system_inop_p

Entered the function propulsion_system_p

Entered the function steering_system_inop_p

Entered the function gps_needed_p

Entered the function reach_waypoint_p

Entered the function diving_system_p

Entered the function bouyancy_system_p

Entered the function thruster_system_p

Entered the function leak_test_p

Entered the function payload_prob_p

Entered the function unknown_obstacle_p

Entered the function send_setpoints_and_modes

Entered the function return_done_p

Entered the function wait_for_recovery

By reaching this primitive goal, the inference engine completes its search. In the case of Prolog, the initial query is satisfied and the system responds with "yes" to indicate that the mission has been successfully completed. In CLIPS, the inference engine halts its search because no rules remain on the agenda.

The next trace is generated from the same rule sets and, hence, represent execution of the same mission. In this case, however, a problem involving the battery is encountered, resulting in a different sequence of primitive goals being generated.

The mission is initiated and launched as before:

Entered the function ready_vehicle_for_launch_p

Entered the function select_first_waypoint

Entered the function in_transit_p

Entered the function power_gone_p

Entered the function computer_system_inop_p

Entered the function propulsion_system_p

Entered the function steering_system_inop_p

Entered the function gps_needed_p

Entered the function reach_waypoint_p

Entered the function diving_system_p

Entered the function bouyancy_system_p

Entered the function thruster_system_p

Entered the function leak_test_p

Entered the function payload_prob_p

Entered the function unknown_obstacle_p

Entered the function send_setpoints_and_modes

Entered the function transit_done_p

Entered the function in_transit_p

Entered the function power_gone_p

Entered the function computer_system_inop_p

At some point during the transit, the battery level drops below the threshold of acceptability and the order to surface is given:

Entered the function computer_system_inop_p

Entered the function propulsion_system_p

Entered the function steering_system_inop_p

Entered the function gps_needed_p

Entered the function reach_waypoint_p

Entered the function diving_system_p

Entered the function bouyancy_system_p

Entered the function thruster_system_p

Entered the function leak_test_p

Entered the function payload_prob_p

Entered the function unknown_obstacle_p

Entered the function send_setpoints_and_modes

Entered the function transit_done_p

Entered the function in_transit_p

Entered the function power_gone_p

Entered the function surface

At this point, control of the vehicle is turned over to the Tactical level. The Tactical level is then charged with issuing the appropriate commands to the Execution level which result in the desired action. For this implementation, once the vehicle reached the surface, communications links between the RBM levels were interrupted and the simulation stopped. Any number of alternatives are certainly possi-

ble, including initiation of a radio link or some other "SOS" signal. The Strategic level can be made to include this reasoning by adding the appropriate rules to its rule base.

APPENDIX B. TACTICAL LEVEL SOURCE CODE

```
-- File: auv_ood_spec.ca
-- Author: Ron B. Byrnes
-- Date: 2 Oct 92
-- Revised: 15 Dec 92
-- System: Grus
-- Compiler: Classic-Ada, VADS
-- Description: Oversees the execution of the Tactical level.
-- Coordinates requests for information between the objects
-- of the object hierarchy. Provides the interface between
-- the Tactical level and the Strategic level.

class AUV_OOD is

  method CREATE (NEW_AUV_OOD : out OBJECT_ID);
  instance method INITIALIZE;
  instance method LINKUP_MM (POINTER : OBJECT_ID);
  instance method LINKUP_SR (POINTER : OBJECT_ID);
  instance method DELETE;
  instance method DOWNLOAD_OBSTACLES;
  instance method DOWNLOAD_INITIAL_STATE;
  instance method DOWNLOAD_WAYPOINTS;
  instance method DOWNLOAD_FINAL_GOAL;
  instance method SEL_1ST_WP;
  instance method TRANSMIT_COMMAND (HEADING : in FLOAT;
    ZWP : in FLOAT; SPEED : in FLOAT; XWP : in FLOAT;
    YWP : in FLOAT; MODE : in INTEGER);
  instance method SYS_CHECK (ANS : out INTEGER);
  instance method POWER_CHECK (ANS : out INTEGER);
  instance method SURFACE;
  instance method REACH_WAYPOINT (ANS : out BOOLEAN);
  instance method GET_NEXT_WP;
  instance method GET_VEH_POSTURE;
  instance method OBS_CHECK (ANS : out BOOLEAN);
  instance method PLAN;
  instance method EXECUTE_PLAN;
  instance method REACH_GOAL (ANS : out BOOLEAN);
  instance method NAV_OOD_BACKLINK(OOD : OBJECT_ID);
  instance method ENGR_OOD_BACKLINK(OOD : OBJECT_ID);
  instance method WEAP_OOD_BACKLINK(OOD : OBJECT_ID);

end AUV_OOD;
```

```
-- File: auv_ood_body.ca
-- Author: Ron B. Byrnes
-- Date: 2 Oct 92
-- Revised: 15 Dec 92
-- System: Grus
-- Compiler: Classic-Ada, VADS
-- Description: Following creation, all dependent objects are
-- created as part of the initialization. Power model implemented,
-- which resides in ENGINEER

with TEXT_IO, NAVIGATOR, ENGINEER, WEAPONS_OFFICER,
   COMMAND_SENDER, C_LIB;
use TEXT_IO, C_LIB;

class body AUV_OOD is

  NAV : instance OBJECT_ID; -- instance of Navigator class
  ENGR : instance OBJECT_ID; -- instance of ENGINEER class
  WEAP : instance OBJECT_ID; -- instance of WEAPONS_OFFICER class
  CMDS : instance OBJECT_ID; -- instance of COMMAND_SENDER class
  MISS_MODEL : instance OBJECT_ID; -- pointer to MISSION_MODEL
  NUM_F, X, Y, Z, X_INIT, Y_INIT, Z_INIT,
  HEAD_INIT, SPEED_VAL, X_WAYPOINT, Y_WAYPOINT,
  Z_WAYPOINT, SPEED_WAYPOINT, CMD_HEADING : instance FLOAT;
  NUM_I, MODE_NUM : instance INTEGER;

  method CREATE (NEW_AUV_OOD : out OBJECT_ID) is
  begin
    NEW_AUV_OOD := instantiate;
    send (NEW_AUV_OOD, INITIALIZE); -- initialize upon creation
    PUT_LINE("AUV_OOD object is created!");
    NEW_LINE;
  end CREATE;

  instance method INITIALIZE is
  begin
    NAV := NAVIGATOR.class_object;
    send(NAV, CREATE, new_navigator => NAV);
    send(NAV, GUI_NAV_BACKLINK, navig => NAV);
    send(NAV, GPS_NAV_BACKLINK, navig => NAV);
    send(NAV, SON_NAV_BACKLINK, navig => NAV);
    send(NAV, DR_NAV_BACKLINK, navig => NAV);
    send(NAV, MR_NAV_BACKLINK, navig => NAV);
    ENGR := ENGINEER.class_object;
    send(ENGR, CREATE, new_engineer => ENGR);
    WEAP := WEAPONS_OFFICER.class_object;
    send(WEAP, CREATE, new_weapons_officer => WEAP);
    CMDS := COMMAND_SENDER.class_object;
    send(CMDS, CREATE, new_command_sender => CMDS);
  end INITIALIZE;
```

```
instance method NAV_OOD_BACKLINK (OOD : OBJECT_ID) is
begin
  PUT_LINE("Backlinking NAVIGATOR and OOD.");
  send(NAV, PARENT_LINK, OOD => OOD);
end NAV_OOD_BACKLINK;

instance method ENGR_OOD_BACKLINK (OOD : OBJECT_ID) is
begin
  PUT_LINE("Backlinking ENGINEER and OOD.");
  send(ENGR, PARENT_LINK, OOD => OOD);
end ENGR_OOD_BACKLINK;

instance method WEAP_OOD_BACKLINK (OOD : OBJECT_ID) is
begin
  PUT_LINE("Backlinking WEAPONS_OFFICER and OOD.");
  send(WEAP, PARENT_LINK, OOD => OOD);
end WEAP_OOD_BACKLINK;

instance method LINKUP_MM (POINTER : OBJECT_ID) is
begin
  MISS_MODEL := POINTER;
  PUT_LINE("Passing ptr to MM to NAV.");
  send(NAV, LINKUP, MM => POINTER);
end LINKUP_MM;

instance method LINKUP_SR (POINTER : OBJECT_ID) is
begin
  PUT_LINE("Passing ptr to SR to NAV.");
  send(NAV, LINK_SR, SR => POINTER);
end LINKUP_SR;

instance method DOWNLOAD_OBSTACLES is
begin
  send(MISS_MODEL, GET_NUM_OBSTACLES, num_obs => NUM_I);
  NUM_F := FLOAT(NUM_I); PUT_LINE(" The number of obstacles is");
  put_float(NUM_F);
  PUT_LINE(" and the obstacles themselves:");
  for I in 1..NUM_I loop
    send(MISS_MODEL, GET_OBSTACLE, index => I, x_coord => X,
      y_coord => Y, z_coord => Z);
    put_float(X);
    put_float(Y);
    put_float(Z);
  end loop;
end DOWNLOAD_OBSTACLES;

instance method DOWNLOAD_INITIAL_STATE is
begin
  send(MISS_MODEL, GET_INITIAL_STATE, init_x => X_INIT,
    init_y => Y_INIT, init_z => Z_INIT,
  init_heading => HEAD_INIT);
```

```
  put_float(X_INIT);
  put_float(Y_INIT);
  put_float(Z_INIT);
  put_float(HEAD_INIT);
end DOWNLOAD_INITIAL_STATE;

instance method DOWNLOAD_WAYPOINTS is
begin
  send(MISS_MODEL, GET_NUM_WAYPOINTS, num_wp => NUM_I);
  NUM_F := FLOAT(NUM_I);
  PUT_LINE(" The number of waypoints is");
  put_float(NUM_F);
  PUT_LINE(" and the waypoints are:");
  for I in 1..NUM_I loop
    send(MISS_MODEL, GET_WAYPOINT, index => I, speed => SPEED_VAL,
      x_coord => X, y_coord => Y, z_coord => Z);
    put_float(SPEED_VAL);
    put_float(X);
    put_float(Y);
    put_float(Z);
  end loop;
end DOWNLOAD_WAYPOINTS;

instance method DOWNLOAD_FINAL_GOAL is
begin
  send(MISS_MODEL, GET_FINAL_GOAL, x_final => X, y_final => Y);
  put_float(X);
  put_float(Y);
end DOWNLOAD_FINAL_GOAL;

instance method SEL_1ST_WP is
begin
  send(NAV, LOAD_INIT_AND_GOAL_POSN);
  send(NAV, LOAD_WP, x_wp => X_WAYPOINT, y_wp => Y_WAYPOINT,
    z_wp => Z_WAYPOINT, sp_wp => SPEED_WAYPOINT);
  send(NAV, GET_HEADING, commanded_heading => CMD_HEADING);
  send(self, TRANSMIT_COMMAND, heading => CMD_HEADING,
    zwp => Z_WAYPOINT, speed => SPEED_WAYPOINT,
    xwp => X_WAYPOINT, ywp => Y_WAYPOINT, mode => MODE_NUM);
  PUT_LINE("SEL_1ST_WP Done!");
  NEW_LINE;
  NEW_LINE;
end SEL_1ST_WP;

instance method TRANSMIT_COMMAND (HEADING : FLOAT; ZWP : FLOAT;
  SPEED : FLOAT; XWP : FLOAT; YWP : FLOAT; MODE : INTEGER) is
begin
  send(CMDS, SEND_COMMAND_PACKET, head_cmd => HEADING,
    z_cmd => ZWP, sp_cmd => SPEED, x_cmd => XWP, y_cmd => YWP,
    mode_cmd => MODE);
end TRANSMIT_COMMAND;
```

```
instance method SYS_CHECK (ANS : out INTEGER) is
begin
  send(ENGR, STATUS_REPORT, report => ANS);
end SYS_CHECK;

instance method POWER_CHECK (ANS : out INTEGER) is
begin
  send(ENGR, POWER_REPORT, report => ANS);
end POWER_CHECK;

instance method SURFACE is
begin
PUT_LINE("Surfacing now...");
  for  I in 1..25 loop
    send(self, GET_VEH_POSTURE);
    send(self, PLAN);
    send(self, TRANSMIT_COMMAND, heading => CMD_HEADING, zwp => 0.0,
    speed => 0.0, xwp => X_WAYPOINT, ywp => Y_WAYPOINT,
    mode => MODE_NUM);
  end loop;
end SURFACE;


instance method REACH_WAYPOINT (ANS : out BOOLEAN) is
begin
  send(NAV, WP_REACHED, result => ANS);
end REACH_WAYPOINT;

 instance method GET_NEXT_WP is
 begin
  send(NAV, LOAD_WP, x_wp => X_WAYPOINT, y_wp => Y_WAYPOINT,
    z_wp => Z_WAYPOINT, sp_wp => SPEED_WAYPOINT);
end GET_NEXT_WP;

instance method GET_VEH_POSTURE is
begin
  send(NAV, RECEIVE_POSN);
  send(NAV, RECEIVE_HEADING);
  send(NAV, RECEIVE_SPEED);
end GET_VEH_POSTURE;

instance method OBS_CHECK (ANS : out BOOLEAN) is
begin
  send(NAV, CHECK_OBSTACLES, report => ANS);
end OBS_CHECK;

instance method PLAN is
begin
  send(NAV, GET_HEADING, commanded_heading => CMD_HEADING);
end PLAN;
```

```
  instance method EXECUTE_PLAN is
  begin
    send(self, TRANSMIT_COMMAND, heading => CMD_HEADING,
      zwp => Z_WAYPOINT, speed => SPEED_WAYPOINT, xwp => X_WAYPOINT,
      ywp => Y_WAYPOINT, mode => MODE_NUM);
  end EXECUTE_PLAN;

  instance method REACH_GOAL (ANS : out BOOLEAN) is
  begin
    send(NAV, CHECK_GOAL, report => ANS);
  end REACH_GOAL;

  instance method DELETE is
  begin
    send(NAV, DELETE);
    send(ENGR, DELETE);
    send(WEAP, DELETE);
    send(CMDS, DELETE);
    PUT_LINE("auv_ood object destroyed!");
    destroy;
  end DELETE;

end AUV_OOD;
```

```
-- File: navigator_spec.ca
-- Author: Ron B. Byrnes
-- Date: 2 Oct 92
-- Revised : 24 Nov 92
-- System: Grus
-- Compiler: Classic-Ada, VADS
-- Description: Responsible for the steering and location of the
-- vehicle.

class NAVIGATOR is

 method CREATE (NEW_NAVIGATOR : out OBJECT_ID);
 instance method INITIALIZE;
 instance method GUI_NAV_BACKLINK (NAVIG : OBJECT_ID);
 instance method GPS_NAV_BACKLINK (NAVIG : OBJECT_ID);
 instance method SON_NAV_BACKLINK (NAVIG : OBJECT_ID);
 instance method DR_NAV_BACKLINK (NAVIG : OBJECT_ID);
 instance method MR_NAV_BACKLINK (NAVIG : OBJECT_ID);
 instance method PARENT_LINK (OOD : OBJECT_ID);
 instance method LINKUP(MM : OBJECT_ID);
 instance method LINK_SR (SR : OBJECT_ID);
 instance method LOAD_INIT_AND_GOAL_POSN;
 instance method LOAD_WP (X_WP : out FLOAT; Y_WP : out FLOAT;
 Z_WP : out FLOAT; SP_WP : out FLOAT);
 instance method GET_HEADING (COMMANDED_HEADING : out FLOAT);

 instance method WP_REACHED (RESULT : out BOOLEAN);
 instance method RECEIVE_POSN;
 instance method RECEIVE_HEADING;
 instance method RECEIVE_SPEED;
 instance method CHECK_OBSTACLES (REPORT : out BOOLEAN);
 instance method CHECK_GOAL (REPORT : out BOOLEAN);

 instance method DELETE;

end NAVIGATOR;
```

```
-- File: navigator_body.ca
-- Author: Ron B. Byrnes
-- Date: 2 Oct 92
-- Revised: 24 Nov 92
-- System: Grus
-- Compiler: Classic-Ada, VADS
-- Description: Instantiates dependent objects upon initialization.

with TEXT_IO, GUIDANCE, GPS_CONTROL, SONAR_CONTROL, DEAD_RECK-
ONING,
 MISSION_REPLANNER, MATH;
use TEXT_IO, MATH;

class body NAVIGATOR is

 package FLOAT_INOUT is new FLOAT_IO(FLOAT);
 use FLOAT_INOUT;

 GUIDE : instance OBJECT_ID; -- Instance of GUIDANCE class
 GPS : instance OBJECT_ID; -- Instance of GPS_CONTROL class
 SONAR : instance OBJECT_ID; -- Instance of SONAR_CONTROL class
 DR : instance OBJECT_ID; -- Instance of DEAD_RECKONING class
 REPLAN : instance OBJECT_ID; -- Instance of MISSION_REPLANNER class
 OOD_HANDLE : instance OBJECT_ID;
 MISS_MODEL : instance OBJECT_ID;
 SENSORY_RECEIVER : instance OBJECT_ID;
 X_POSITION, Y_POSITION, Z_POSITION,
 X_WAYPOINT, Y_WAYPOINT, Z_WAYPOINT, X_GOAL, Y_GOAL,
 UNNEEDED, CALC, WP_THRESHOLD,
 TRUE_SPEED, TRUE_HEADING : instance FLOAT;

 WP_INDEX : instance INTEGER;


 method CREATE (NEW_NAVIGATOR : out OBJECT_ID) is
 begin
 NEW_NAVIGATOR := instantiate;
 send (NEW_NAVIGATOR, INITIALIZE); -- initialize upon creation
 PUT_LINE("Navigator object is created!");
 NEW_LINE;
 end CREATE;

 instance method INITIALIZE is
 begin
 GUIDE := GUIDANCE.class_object;
 send(GUIDE, CREATE, new_guidance => GUIDE);
 send(GUIDE, LOS_GUI_BACKLINK, pointer => GUIDE);
 GPS := GPS_CONTROL.class_object;
 send(GPS, CREATE, new_gps_control => GPS);
 SONAR := SONAR_CONTROL.class_object;
 send(SONAR, CREATE, new_sonar_control => SONAR);
```

```
DR := DEAD_RECKONING.class_object;
send(DR, CREATE, new_dead_reckoning => DR);
REPLAN := MISSION_REPLANNER.class_object;
send(REPLAN, CREATE, new_mission_replanner => REPLAN);
WP_INDEX := 0;
WP_THRESHOLD := 20.0;
end INITIALIZE;

instance method GUI_NAV_BACKLINK (NAVIG : OBJECT_ID) is
begin
PUT_LINE("Backlinking GUIDANCE and NAVIGATOR.");
send(GUIDE, PARENT_LINK, pointer_gn => NAVIG);
end GUI_NAV_BACKLINK;

instance method GPS_NAV_BACKLINK (NAVIG : OBJECT_ID) is
begin
PUT_LINE("Backlinking GPS_CONTROL and NAVIGATOR.");
send(GPS, PARENT_LINK, pointer_gpsn => NAVIG);
end GPS_NAV_BACKLINK;

instance method SON_NAV_BACKLINK (NAVIG : OBJECT_ID) is
begin
PUT_LINE("Backlinking SONAR_CONTROL and NAVIGATOR.");
send(SONAR, PARENT_LINK, pointer_sn => NAVIG);
end SON_NAV_BACKLINK;

instance method DR_NAV_BACKLINK (NAVIG : OBJECT_ID) is
begin
PUT_LINE("Backlinking DEAD_RECKONING and NAVIGATOR.");
send(DR, PARENT_LINK, pointer_dr => NAVIG);
end DR_NAV_BACKLINK;

instance method MR_NAV_BACKLINK (NAVIG : OBJECT_ID) is
begin
PUT_LINE("Backlinking MISSION_REPLANNER and NAVIGATOR.");
send(REPLAN, PARENT_LINK, pointer_mn => NAVIG);
end MR_NAV_BACKLINK;

instance method PARENT_LINK (OOD : OBJECT_ID) is
begin
PUT_LINE("OOD reached NAVIG.");
OOD_HANDLE := OOD;
end PARENT_LINK;

instance method LINKUP(MM : OBJECT_ID) is
begin
PUT_LINE("I'm passing MM to GUIDANCE.");
send(GUIDE, LINKUP, miss_model => MM);
MISS_MODEL := MM;
end LINKUP;
```

```
instance method LINK_SR (SR : OBJECT_ID) is
begin
SENSORY_RECEIVER := SR;
PUT_LINE("Nav got ptr to SR.");
end LINK_SR;

instance method LOAD_INIT_AND_GOAL_POSN is
begin
send(MISS_MODEL, GET_INITIAL_STATE, init_x => X_POSITION,
    init_y => Y_POSITION, init_z => Z_POSITION, init_heading => UNNEEDED);
send(MISS_MODEL, GET_FINAL_GOAL, x_final => X_GOAL, y_final => Y_GOAL);
end LOAD_INIT_AND_GOAL_POSN;

instance method LOAD_WP (X_WP : out FLOAT; Y_WP : out FLOAT;
Z_WP : out FLOAT; SP_WP : out FLOAT) is
begin
WP_INDEX := WP_INDEX + 1;
send(MISS_MODEL, GET_WAYPOINT, index => WP_INDEX, speed => SP_WP,
    x_coord => X_WP, y_coord => Y_WP, z_coord => Z_WP);
X_WAYPOINT := X_WP;
Y_WAYPOINT := Y_WP;
Z_WAYPOINT := Z_WP;
end LOAD_WP;

instance method GET_HEADING (COMMANDED_HEADING : out FLOAT) is
begin
send(GUIDE, GET_HEADING, calculated_heading => COMMANDED_HEADING,
   x_posn => X_POSITION, y_posn => Y_POSITION, z_posn => Z_POSITION,
   x_waypt => X_WAYPOINT, y_waypt => Y_WAYPOINT,
   z_waypt => Z_WAYPOINT);
end GET_HEADING;


instance method WP_REACHED (RESULT : out BOOLEAN) is
begin
CALC := SQRT((X_WAYPOINT - X_POSITION)**2 +
(Y_WAYPOINT - Y_POSITION)**2);
if CALC < WP_THRESHOLD then
RESULT := TRUE;
PUT("Waypoint reached is (X,Y,Z): ");
PUT(X_WAYPOINT, fore => 5, aft => 2, exp => 0);
PUT(Y_WAYPOINT, fore => 5, aft => 2, exp => 0);
PUT(Z_WAYPOINT, fore => 5, aft => 2, exp => 0);
NEW_LINE;
Put("Actual X, Y, depth, and heading :");
PUT(X_POSITION, fore => 5, aft => 2, exp => 0);
PUT(Y_POSITION, fore => 5, aft => 2, exp => 0);
PUT(Z_POSITION, fore => 5, aft => 2, exp => 0);
PUT(TRUE_HEADING, fore => 5, aft => 2, exp => 0);
NEW_LINE;
else
```

```
RESULT := FALSE;
end if;
end WP_REACHED;

instance method RECEIVE_POSN is
begin
send(SENSORY_RECEIVER, GET_CURRENT_POSN, x => X_POSITION,
    y => Y_POSITION, z => Z_POSITION);
end RECEIVE_POSN;

instance method RECEIVE_HEADING is
begin
send(SENSORY_RECEIVER, GET_CURRENT_HEADING,
   head => TRUE_HEADING);
end RECEIVE_HEADING;

instance method RECEIVE_SPEED is
begin
send(SENSORY_RECEIVER, GET_CURRENT_SPEED, speed => TRUE_SPEED);
end RECEIVE_SPEED;

instance method CHECK_OBSTACLES (REPORT : out BOOLEAN) is
begin
send(SONAR, OBSTACLE_STATUS, status => REPORT);
end CHECK_OBSTACLES;

instance method CHECK_GOAL (REPORT : out BOOLEAN) is
begin
CALC := SQRT((X_GOAL - X_POSITION)**2 +
(Y_GOAL - Y_POSITION)**2);
if CALC < WP_THRESHOLD then
REPORT := TRUE;
PUT("Final Goal reached is (X,Y,Z): ");
PUT(X_GOAL, fore => 5, aft => 2, exp => 0);
PUT(Y_GOAL, fore => 5, aft => 2, exp => 0);
NEW_LINE;
Put("Actual X, Y, depth, and heading :");
PUT(X_POSITION, fore => 5, aft => 2, exp => 0);
PUT(Y_POSITION, fore => 5, aft => 2, exp => 0);
PUT(Z_POSITION, fore => 5, aft => 2, exp => 0);
PUT(TRUE_HEADING, fore => 5, aft => 2, exp => 0);
NEW_LINE;

else
REPORT := FALSE;
end if;
end CHECK_GOAL;


instance method DELETE is
```

```
begin
send(GUIDE, DELETE);
send(GPS, DELETE);
send(SONAR, DELETE);
send(DR, DELETE);
send(REPLAN, DELETE);
PUT_LINE("Navigator object going down...!");
destroy;
end DELETE;

end NAVIGATOR;
```

```
-- File: guidance_spec.ca
-- Author: Ron B. Byrnes
-- Date: 2 Oct 92
-- Revised: 16 Nov 92
-- System: Grus
-- Compiler: Classic-Ada, VADS
-- Description: Responsible for the calculation of heading
-- commands.

class GUIDANCE is

 method CREATE (NEW_GUIDANCE : out OBJECT_ID);
 instance method INITIALIZE;
 instance method PARENT_LINK (POINTER_GN : OBJECT_ID);
 instance method LINKUP (MISS_MODEL : OBJECT_ID);
 instance method LOS_GUI_BACKLINK (POINTER : OBJECT_ID);
 instance method GET_HEADING (CALCULATED_HEADING : out FLOAT;
     X_POSN, Y_POSN, Z_POSN, X_WAYPT, Y_WAYPT, Z_WAYPT : in FLOAT);
 instance method DELETE;

end GUIDANCE;
```

```
-- File: guidance_body.ca
-- Author: Ron B. Byrnes
-- Date: 2 Oct 92
-- Revised: 16 Nov 92
-- System: Grus
-- Compiler: Classic-Ada, VADS
-- Description: Instantiates "dependent" objects upon initialization.

with TEXT_IO, LOS_CALCULATOR;
use TEXT_IO;

class body GUIDANCE is

 LOS : instance OBJECT_ID;
 NAVIG_HANDLE : instance OBJECT_ID;

 method CREATE (NEW_GUIDANCE : out OBJECT_ID) is
 begin
 NEW_GUIDANCE := instantiate;
 send (NEW_GUIDANCE, INITIALIZE); -- intialize upon creation
 PUT_LINE("Guidance object is created!");
 NEW_LINE;
 end CREATE;

 instance method INITIALIZE is
 begin
 LOS := LOS_CALCULATOR.class_object;
 send(LOS, CREATE, new_los_calculator => LOS);
 end INITIALIZE;

 instance method GET_HEADING (CALCULATED_HEADING : out FLOAT;
 X_POSN, Y_POSN, Z_POSN, X_WAYPT, Y_WAYPT, Z_WAYPT : in FLOAT) is
 begin
 send(LOS, GET_NEW_HEADING, heading_set_point => CALCULATED_HEADING,
     xcurr => X_POSN, ycurr => Y_POSN, zcurr => Z_POSN,
     xnext => X_WAYPT, ynext => Y_WAYPT, znext => Z_WAYPT);
 end GET_HEADING;

 instance method PARENT_LINK (POINTER_GN : OBJECT_ID) is
 begin
 PUT_LINE("NAVIG reached GUIDE.");
 NAVIG_HANDLE := POINTER_GN;
 end PARENT_LINK;

 instance method LOS_GUI_BACKLINK(POINTER : OBJECT_ID) is
 begin
 PUT_LINE("Backlinking LOS and GUIDANCE.");
 send(LOS, PARENT_LINK, guide => POINTER);
 end LOS_GUI_BACKLINK;

 instance method LINKUP(MISS_MODEL : OBJECT_ID) is
```

```
begin
PUT_LINE("I'm passing ptr to MISS_MODEL to LOS.");
send(LOS, LINKUP, MM => MISS_MODEL);
end LINKUP;

instance method DELETE is
begin
send(LOS, DELETE);
PUT_LINE("Guidance object going down!");
destroy;
end DELETE;

end GUIDANCE;
```

```
-- File: gps_control_spec.ca
-- Author: Ron B. Byrnes
-- Date: 5 Oct 92
-- Revised 25 Nov 92
-- System: Grus
-- Compiler: Classic-Ada, VADS
-- Description: Generates commands to activate/deactivate gps
-- package. Obtains and analyzes readings. Provides positional
-- information derived from those readings.

class GPS_CONTROL is

 method CREATE (NEW_GPS_CONTROL : out OBJECT_ID);
 instance method INITIALIZE;
 instance method PARENT_LINK (POINTER_GPSN : OBJECT_ID);
 instance method DELETE;

end GPS_CONTROL;
```

```
-- File: gps_control_body.ca
-- Author: Ron B. Byrnes
-- Date: 5 Oct 92
-- Revised: 25 Nov 92
-- System: Grus
-- Compiler: Classic-Ada, VADS
-- Description:

with TEXT_IO;
use TEXT_IO;

class body GPS_CONTROL is

 NAVIG_HANDLE : instance OBJECT_ID;

 method CREATE (NEW_GPS_CONTROL : out OBJECT_ID) is
 begin
 NEW_GPS_CONTROL := instantiate;
 send (NEW_GPS_CONTROL, INITIALIZE); -- initialize upon creation
 end CREATE;

 instance method INITIALIZE is
 begin
 PUT_LINE("GPS Control instantiated.");
 end INITIALIZE;

 instance method PARENT_LINK (POINTER_GPSN : OBJECT_ID) is
 begin
 PUT_LINE("NAVIG reached GPS.");
 NAVIG_HANDLE := POINTER_GPSN;
 end PARENT_LINK;

 instance method DELETE is
 begin
 PUT_LINE("GPS Control being deallocated now.");
 destroy;
 end DELETE;

end GPS_CONTROL;
```

```
-- File: sonar_control_spec.ca
-- Author: Ron B. Byrnes
-- Date: 5 Oct 92
-- Revised: 25 Nov 92
-- System: Grus
-- Compiler: Classic-Ada, VADS
-- Description: Activates/deactivates sonar(s), takes readings
-- from sonars and determines location and identification of
-- objects; provides vehicle position in conjunction with
-- the world model.

class SONAR_CONTROL is

 method CREATE (NEW_SONAR_CONTROL : out OBJECT_ID);
 instance method INITIALIZE;
 instance method PARENT_LINK (POINTER_SN : OBJECT_ID);
 instance method OBSTACLE_STATUS (STATUS : out BOOLEAN);
 instance method DELETE;

end SONAR_CONTROL;
```

```
-- File: sonar_control_body.ca
-- Author: Ron B. Byrnes
-- Date: 5 Oct 92
-- Revised 25 Nov 92
-- System: Grus
-- Compiler: Classic-Ada, VADS
-- Description:

with TEXT_IO;
use TEXT_IO;

class body SONAR_CONTROL is

 NAVIG_HANDLE : instance OBJECT_ID;

 method CREATE (NEW_SONAR_CONTROL : out OBJECT_ID) is
 begin
 NEW_SONAR_CONTROL := instantiate;
 send (NEW_SONAR_CONTROL, INITIALIZE); -- initialize upon creation
 end CREATE;

 instance method INITIALIZE is
 begin
 PUT_LINE("Sonar Controller has been instantiated.");
 end INITIALIZE;

 instance method PARENT_LINK (POINTER_SN : OBJECT_ID) is
 begin
 PUT_LINE("NAVIG reached SONAR.");
 NAVIG_HANDLE := POINTER_SN;
 end PARENT_LINK;

 instance method OBSTACLE_STATUS (STATUS : out BOOLEAN) is
 begin
 STATUS := FALSE; -- Obstacle identification algorithm goes here
 end OBSTACLE_STATUS;

 instance method DELETE is
 begin
 PUT_LINE("Sonar control going down.");
 destroy;
 end DELETE;

end SONAR_CONTROL;
```

```
-- File: dead_reckoning_spec.ca
-- Author: Ron B. Byrnes
-- Date: 5 Oct 92
-- Revised: 24 Nov 92
-- System: Grus
-- Compiler: Classic-Ada, VADS
-- Description: Determines vehicle position by dead reckoning

class DEAD_RECKONING is

 method CREATE (NEW_DEAD_RECKONING : out OBJECT_ID);
 instance method INITIALIZE;
 instance method PARENT_LINK (POINTER_DR : OBJECT_ID);
 instance method DELETE;

end DEAD_RECKONING;
```

```
-- File: dead_reckoning_body.ca
-- Author: Ron B. Byrnes
-- Date: 5 Oct 92
-- Revised: 24 Nov 92
-- System: Grus
-- Compiler: Classic-Ada, VADS
-- Description: Will estimate vehicle location in the X-Y
-- plane using dead reckoning. Required parameters
-- will be obtained from the Sensory Receiver (the
-- link to which needs to be established).

with TEXT_IO;
use TEXT_IO;

class body DEAD_RECKONING is

 NAVIG_HANDLE : instance OBJECT_ID;

 method CREATE (NEW_DEAD_RECKONING : out OBJECT_ID) is
 begin
 NEW_DEAD_RECKONING := instantiate;
 send (NEW_DEAD_RECKONING, INITIALIZE); -- initialize upon creation
 end CREATE;

 instance method INITIALIZE is
 begin
 PUT_LINE("Dead Reckoner object created.");
 end INITIALIZE;

 instance method PARENT_LINK (POINTER_DR : OBJECT_ID) is
 begin
 PUT_LINE("NAVIG reached DR.");
 NAVIG_HANDLE := POINTER_DR;
 end PARENT_LINK;

 instance method DELETE is
 begin
 PUT_LINE("Dead Reckoner being destroyed.");
 destroy;
 end DELETE;

end DEAD_RECKONING;
```

```
-- File: mission_replanner_spec.ca
-- Author: Ron B. Byrnes
-- Date: 5 Oct 92
-- Revised: 25 Nov 92
-- System: Grus
-- Compiler: Classic-Ada, VADS
-- Description: Performs local replanning (obstacle avoidance) and
-- global replanning (fault tolerance) as directed.

class MISSION_REPLANNER is

 method CREATE (NEW_MISSION_REPLANNER : out OBJECT_ID);
 instance method INITIALIZE;
 instance method PARENT_LINK (POINTER_MN : OBJECT_ID);
 instance method DELETE;

end MISSION_REPLANNER;
```

```
-- File: mission_replanner_body.ca
-- Author: Ron B. Byrnes
-- Date: 5 Oct 92
-- Revised: 25 Nov 92
-- System: Grus
-- Compiler: Classic-Ada, VADS
-- Description:

with TEXT_IO;
use TEXT_IO;

class body MISSION_REPLANNER is

 NAVIG_HANDLE : instance OBJECT_ID;

 method CREATE (NEW_MISSION_REPLANNER : out OBJECT_ID) is
 begin
 NEW_MISSION_REPLANNER := instantiate;
 send (NEW_MISSION_REPLANNER, INITIALIZE); -- initialize upon creation
 end CREATE;

 instance method INITIALIZE is
 begin
 PUT_LINE("Mission Replanner object created.");
 end INITIALIZE;

 instance method PARENT_LINK (POINTER_MN : OBJECT_ID) is
 begin
 PUT_LINE("NAVIG reached REPLAN.");
 NAVIG_HANDLE := POINTER_MN;
 end PARENT_LINK;

 instance method DELETE is
 begin
 PUT_LINE("Mission Replanner going down.");
 destroy;
 end DELETE;

end MISSION_REPLANNER;
```

```
-- File: engineer_spec.ca
-- Author: Ron B. Byrnes
-- Date: 5 Oct 92
-- Revised: 15 Dec 92
-- System: Grus
-- Compiler: Classic-Ada, VADS
-- Description:

class ENGINEER is

 method CREATE (NEW_ENGINEER : out OBJECT_ID);
 instance method INITIALIZE;
 instance method PARENT_LINK(OOD : OBJECT_ID);
 instance method STATUS_REPORT (REPORT : out INTEGER);
 instance method POWER_REPORT (REPORT : out INTEGER);
 instance method DELETE;

end ENGINEER;
```

```
-- File: engineer_body.ca
-- Author: Ron B. Byrnes
-- Date: 5 Oct 92
-- Revised: 15 Dec 92
-- System: Grus
-- Compiler: Classic-Ada, VADS
-- Description: Responsible for the monitoring and health of the vehicle's
-- subsystems. Battery model resides here.

with TEXT_IO;
use TEXT_IO;


class body ENGINEER is

 OOD_HANDLE : instance OBJECT_ID;

 BATTERY : instance FLOAT;

 method CREATE (NEW_ENGINEER : out OBJECT_ID) is
 begin
 NEW_ENGINEER := instantiate;
 send (NEW_ENGINEER, INITIALIZE); -- initialize upon creation
 end CREATE;

 instance method INITIALIZE is
 begin
 BATTERY := 24.0;
 PUT_LINE("Engineer has been created.");
 end INITIALIZE;

 instance method PARENT_LINK (OOD : OBJECT_ID) is
 begin
 PUT_LINE("OOD reached ENGR.");
 OOD_HANDLE := OOD;
 end PARENT_LINK;

 instance method STATUS_REPORT (REPORT : out INTEGER) is
 begin
 -- read and compare various systems parameters
-- Devise a listing of codes, each associated with a particular system fault.
REPORT := 1;
 end STATUS_REPORT;

 instance method POWER_REPORT (REPORT : out INTEGER) is
 begin
 BATTERY := BATTERY - 0.08;
 if BATTERY < 18.0 then
 REPORT := 0;
 else
 REPORT := 1;
```

```
end if;
end POWER_REPORT;

instance method DELETE is
begin
PUT_LINE("Engineer being destroyed");
destroy;
end DELETE;

end ENGINEER;
```

```
-- File: weapons_officer_spec.ca
-- Author: Ron B. Byrnes
-- Date: 5 Oct 92
-- Revised: 16 Nov 92
-- System: Grus
-- Compiler: Classic-Ada, VADS
-- Description: Responsible for the deployment of the vehicle's
-- payload(s).

class WEAPONS_OFFICER is

 method CREATE (NEW_WEAPONS_OFFICER: out OBJECT_ID);
 instance method INITIALIZE;
 instance method PARENT_LINK(OOD : OBJECT_ID);
 instance method DELETE;

end WEAPONS_OFFICER;
```

```
-- File: weapons_officer_body.ca
-- Author: Ron B. Byrnes
-- Date: 5 Oct 92
-- Revised 16 Nov 92
-- System: Grus
-- Compiler: Classic-Ada, VADS
-- Description:

with TEXT_IO;
use TEXT_IO;

class body WEAPONS_OFFICER is

 OOD_HANDLE : instance OBJECT_ID;

 method CREATE (NEW_WEAPONS_OFFICER : out OBJECT_ID) is
 begin
 NEW_WEAPONS_OFFICER := instantiate;
 send (NEW_WEAPONS_OFFICER, INITIALIZE); -- initialize upon creation
 end CREATE;

 instance method INITIALIZE is
 begin
 PUT_LINE("Weapons Officer created.");
 end INITIALIZE;

 instance method PARENT_LINK (OOD : OBJECT_ID) is
 begin
 PUT_LINE("OOD reached WEAPONS.");
 OOD_HANDLE := OOD;
 end PARENT_LINK;

 instance method DELETE is
 begin
 PUT_LINE("Weapons officer destroyed.");
 destroy;
 end DELETE;

end WEAPONS_OFFICER;
```

```
-- File: command_sender_spec.ca
-- Author: Ron B. Byrnes
-- Date: 5 Oct 92
-- Revised: 23 Nov 92
-- System: Grus
-- Compiler: Classic-Ada, VADS
-- Description: Collects all commands (setpoints) pertaining to the
-- current mode and needed to drive the autopilots located in the
-- Execution level. Builds a command packet using these individual
-- commands and sends the packet to the command port.

class COMMAND_SENDER is

 method CREATE (NEW_COMMAND_SENDER : out OBJECT_ID);
 instance method INITIALIZE;
 instance method SEND_COMMAND_PACKET (HEAD_CMD : in FLOAT;
     Z_CMD : in FLOAT; SP_CMD : in FLOAT; X_CMD : in FLOAT;
     Y_CMD : in FLOAT; MODE_CMD : in INTEGER);
 instance method DELETE;

end COMMAND_SENDER;
```

```
-- File: command_sender_body.ca
-- Author: Ron B. Byrnes
-- Date: 5 Oct 92
-- Revised: 13 Dec 92
-- System: Grus
-- Compiler: Classic-Ada, VADS
-- Description:


with TEXT_IO, C_LIB;
use TEXT_IO, C_LIB;

class body COMMAND_SENDER is

 package FLOAT_INOUT is new FLOAT_IO(FLOAT);
 use FLOAT_INOUT;

 method CREATE (NEW_COMMAND_SENDER : out OBJECT_ID) is
 begin
 NEW_COMMAND_SENDER := instantiate;
 send (NEW_COMMAND_SENDER, INITIALIZE);
 end CREATE;

 instance method INITIALIZE is
 begin
 PUT_LINE("Command sender has been created and initialized.");
 end INITIALIZE;

 instance method SEND_COMMAND_PACKET (HEAD_CMD : in FLOAT;
     Z_CMD : in FLOAT; SP_CMD : in FLOAT; X_CMD : in FLOAT;
     Y_CMD : in FLOAT; MODE_CMD : in INTEGER) is
 begin
 put_float(HEAD_CMD);
 put_float(Z_CMD);
 put_float(SP_CMD);
 put_float(X_CMD);
 put_float(Y_CMD);
 put_mode(MODE_CMD);
 end SEND_COMMAND_PACKET;

 instance method DELETE is
 begin
 PUT_LINE("Command Sender is going down.");
 destroy;
 end DELETE;

end COMMAND_SENDER;
```

```
-- File: sensory_receiver_spec.ca
-- Author: Ron B. Byrnes
-- Date: 8 Oct 92
-- Revised 25 Nov 92
-- System: Grus
-- Compiler: Classic Ada, VADS
-- Description: Accepts telemetry from Execution level; breaks out
-- individual sensor readings from packet and stores in locations
-- accessable by using objects

class SENSORY_RECEIVER is

 method CREATE (NEW_SENSORY_RECEIVER: out OBJECT_ID);
 instance method INITIALIZE;
 instance method LINKUP_DR (DR : in OBJECT_ID);
 instance method GET_CURRENT_POSN (X, Y, Z : out FLOAT);
 instance method GET_CURRENT_HEADING (HEAD : out FLOAT);
 instance method GET_CURRENT_SPEED (SPEED : out FLOAT);
 instance method DELETE;

end SENSORY_RECEIVER;
```

```
-- File: sensory_receiver_body.ca
-- Author: Ron B. Byrnes
-- Date: 8 Oct 92
-- Revised: 12 Jan 93
-- System: Grus
-- Compiler: Classic Ada, VADS
-- Description: Defines the external interface of the object

with TEXT_IO, C_LIB;
use TEXT_IO, C_LIB;

class body SENSORY_RECEIVER is

 DATA_REC : instance OBJECT_ID; -- pointer to DATA RECORDER
 ALT : instance FLOAT;

 package FLOAT_INOUT is new FLOAT_IO(FLOAT);
 use FLOAT_INOUT;

 method CREATE (NEW_SENSORY_RECEIVER : out OBJECT_ID) is
 begin
 NEW_SENSORY_RECEIVER := instantiate;
 send (NEW_SENSORY_RECEIVER, INITIALIZE);
 end CREATE;

 instance method INITIALIZE is
 begin
 PUT_LINE("Sensory Receiver created.");
 end INITIALIZE;

 instance method LINKUP_DR (DR : in OBJECT_ID) is
 begin
 DATA_REC := DR;
 PUT_LINE("SR knows about DR.");
 end LINKUP_DR;

 instance method GET_CURRENT_POSN (X, Y, Z : out FLOAT) is
 begin
 X := get_float;
 Y := get_float;
 ALT := get_float;
 Z := get_float;
 end GET_CURRENT_POSN;

 instance method GET_CURRENT_HEADING (HEAD : out FLOAT) is
 begin
 HEAD := get_float;
 end GET_CURRENT_HEADING;

 instance method GET_CURRENT_SPEED (SPEED : out FLOAT) is
 begin
```

```
SPEED := 0.0; -- Speed isn't obtained from this version of the auvsim
end GET_CURRENT_SPEED;


instance method DELETE is
begin
PUT_LINE("Sensory receiver destroyed.");
destroy;
end DELETE;

end SENSORY_RECEIVER;
```

```
-- File: mission_model_spec.ca
-- Author: Ron B. Byrnes
-- Date: 8 Oct 92
-- Revised: 23 Nov 92
-- System: Grus
-- Compiler: Classic Ada, VADS
-- Description: A data base containing the various mission parameters

class MISSION_MODEL is

 method CREATE (NEW_MISSION_MODEL: out OBJECT_ID);
 instance method INITIALIZE;
 instance method GET_NUM_OBSTACLES (NUM_OBS : out INTEGER);
 instance method GET_OBSTACLE (INDEX : in INTEGER;
 X_COORD : out FLOAT; Y_COORD : out FLOAT;
 Z_COORD : out FLOAT);
 instance method GET_INITIAL_STATE (INIT_X : out FLOAT;
 INIT_Y : out FLOAT; INIT_Z : out FLOAT;
 INIT_HEADING : out FLOAT);

 instance method GET_NUM_WAYPOINTS (NUM_WP : out INTEGER);
 instance method GET_WAYPOINT (INDEX : in INTEGER;
 SPEED : out FLOAT; X_COORD : out FLOAT;
 Y_COORD : out FLOAT; Z_COORD : out FLOAT);
 instance method GET_FINAL_GOAL (X_FINAL : out FLOAT;
 Y_FINAL : out FLOAT);
 instance method GET_NEXT_WP (XWP : out FLOAT; YWP : out FLOAT);
 instance method DELETE;

end MISSION_MODEL;
```

```
-- File: mission_model_body.ca
-- Author: Ron B. Byrnes
-- Date: 8 Oct 92
-- Revised: 15 Dec 92
-- System: Grus
-- Compiler: Classic Ada, VADS
-- Description:

with TEXT_IO;
use TEXT_IO;

class body MISSION_MODEL is

 OBSTACLE_LIST : ARRAY (1..4, 1..3) of FLOAT :=
 ((350.0, 350.0, 50.0),
 (70.0, 1330.0, 50.0),
 (630.0, 1330.0, 50.0),
 (350.0, 1050.0, 50.0));

 -- Waypoints are arranged (speed, x, y, z)
 WAYPOINT_LIST : ARRAY (1..12, 1..4) of FLOAT :=
 (( 300.0, 350.0, 100.0, 10.0),
 ( 300.0, 500.0, 250.0, 30.0),
 ( 300.0, 500.0, 500.0, 35.0),
 ( 300.0, 350.0, 700.0, 40.0),
 ( 300.0, 200.0, 900.0, 40.0),
 ( 300.0, 200.0, 1150.0, 12.0),
 ( 300.0, 350.0, 1300.0, 20.0),
 ( 300.0, 500.0, 1150.0, 30.0),
 ( 300.0, 500.0, 900.0, 40.0),
 ( 300.0, 350.0, 700.0, 50.0),
 ( 300.0, 200.0, 500.0, 50.0),
 ( 300.0, 250.0, 150.0, 50.0));

 I_WP : instance INTEGER;
 NUMBER_OBSTACLES, NUMBER_WAYPOINTS : instance INTEGER;
 INITIAL_X, INITIAL_Y, INITIAL_Z, INITIAL_HEADING,
 FINAL_GOAL_X, FINAL_GOAL_Y : instance FLOAT;

 method CREATE (NEW_MISSION_MODEL : out OBJECT_ID) is
 begin
 NEW_MISSION_MODEL := instantiate;
 send (NEW_MISSION_MODEL, INITIALIZE);
 end CREATE;

 instance method INITIALIZE is
 begin
 I_WP := 0; -- Index to keep track of current waypoint
 NUMBER_OBSTACLES := 4;
 NUMBER_WAYPOINTS := 12;
 INITIAL_X := 0.0;
```

```
INITIAL_Y := 0.0;
INITIAL_Z := 0.0;
INITIAL_HEADING := 90.0;
FINAL_GOAL_X := 250.0;
FINAL_GOAL_Y := 150.0;
PUT_LINE("Mission Model Created.");
end INITIALIZE;

instance method GET_NUM_OBSTACLES (NUM_OBS : out INTEGER) is
begin
NUM_OBS := NUMBER_OBSTACLES;
end GET_NUM_OBSTACLES;

instance method GET_OBSTACLE (INDEX : in INTEGER;
X_COORD : out FLOAT; Y_COORD : out FLOAT;
Z_COORD : out FLOAT) is
begin
X_COORD := OBSTACLE_LIST(INDEX,1);
Y_COORD := OBSTACLE_LIST(INDEX,2);
Z_COORD := OBSTACLE_LIST(INDEX,3);
end GET_OBSTACLE;

instance method GET_INITIAL_STATE (INIT_X : out FLOAT;
INIT_Y : out FLOAT; INIT_Z : out FLOAT;
INIT_HEADING : out FLOAT) is
begin
INIT_X := INITIAL_X;
INIT_Y := INITIAL_Y;
INIT_Z := INITIAL_Z;
INIT_HEADING := INITIAL_HEADING;
end GET_INITIAL_STATE;

instance method GET_NUM_WAYPOINTS (NUM_WP : out INTEGER) is
begin
NUM_WP := NUMBER_WAYPOINTS;
end GET_NUM_WAYPOINTS;

instance method GET_WAYPOINT (INDEX : in INTEGER;
SPEED : out FLOAT; X_COORD : out FLOAT;
Y_COORD : out FLOAT; Z_COORD : out FLOAT) is
begin
SPEED := WAYPOINT_LIST(INDEX,1);
X_COORD := WAYPOINT_LIST(INDEX,2);
Y_COORD := WAYPOINT_LIST(INDEX,3);
Z_COORD := WAYPOINT_LIST(INDEX,4);
end GET_WAYPOINT;

instance method GET_FINAL_GOAL (X_FINAL : out FLOAT;
Y_FINAL : out FLOAT) is
begin
X_FINAL := FINAL_GOAL_X;
```

```
Y_FINAL := FINAL_GOAL_Y;
end GET_FINAL_GOAL;

instance method GET_NEXT_WP (XWP: out FLOAT; YWP : out FLOAT) is
begin
I_WP := I_WP + 1;
XWP := WAYPOINT_LIST(I_WP,2);
YWP := WAYPOINT_LIST(I_WP,3);
end GET_NEXT_WP;

instance method DELETE is
begin
PUT_LINE("Mission Model Destroyed");
destroy;
end DELETE;

end MISSION_MODEL;
```

```
-- File: world_model_spec.ca
-- Author: Ron B. Byrnes
-- Date: 8 Oct 92
-- System: Grus
-- Compiler: Classic Ada, VADS
-- Description: Contains the representation of the known world
-- relavent to the current mission

class WORLD_MODEL is

 method CREATE (NEW_WORLD_MODEL: out OBJECT_ID);
 instance method INITIALIZE;
 instance method DELETE;

end WORLD_MODEL;
```

```
-- File: world_model_body.ca
-- Author: Ron B. Byrnes
-- Date: 8 Oct 92
-- System: Grus
-- Compiler: Classic Ada, VADS
-- Description:

with TEXT_IO;
use TEXT_IO;

class body WORLD_MODEL is

 method CREATE (NEW_WORLD_MODEL : out OBJECT_ID) is
 begin
 NEW_WORLD_MODEL := instantiate;
 send (NEW_WORLD_MODEL, INITIALIZE);
 end CREATE;

 instance method INITIALIZE is
 begin
 PUT_LINE("World Model being created.");
 end INITIALIZE;

 instance method DELETE is
 begin
 PUT_LINE("World Model being destroyed.");
 destroy;
 end DELETE;

end WORLD_MODEL;
```

```
-- File: data_recorder_spec.ca
-- Author: Ron B. Byrnes
-- Date: 8 Oct 92
-- System: Grus
-- Compiler: Classic Ada, VADS
-- Description: Accepts as input telemetry data for post-mission analysis
-- and messages indicating special circumstances (system faults,
-- replans).

class DATA_RECORDER is

 method CREATE (NEW_DATA_RECORDER: out OBJECT_ID);
 instance method INITIALIZE;
 instance method DELETE;

end DATA_RECORDER;
```

```
-- File: data_recorder_body.ca
-- Author: Ron B. Byrnes
-- Date: 8 Oct 92
-- System: Grus
-- Compiler: Classic_Ada, VADS
-- Description:

with TEXT_IO;
use TEXT_IO;

class body DATA_RECORDER is

 method CREATE (NEW_DATA_RECORDER : out OBJECT_ID) is
 begin
 NEW_DATA_RECORDER := instantiate;
 send (NEW_DATA_RECORDER, INITIALIZE);
 end CREATE;

 instance method INITIALIZE is
 begin
 PUT_LINE("Data Recorder created.");
 end INITIALIZE;

 instance method DELETE is
 begin
 PUT_LINE("Data Recorder destroyed.");
 destroy;
 end DELETE;

end DATA_RECORDER;
```

```
-- File: tactical5.ca
-- Author: Ron B. Byrnes
-- Date: 2 Oct 92
-- Revised: 15 Dec 92
-- System: Grus
-- Compiler: Classic_Ada, VADS
-- Description: The program creates the AUV_OOD of the tactical level. Also provides
-- the communications link between the procedural and object-oriented worlds

with TEXT_IO, AUV_OOD, DATA_RECORDER, MISSION_MODEL,
  SENSORY_RECEIVER, WORLD_MODEL, C_LIB, CALENDAR;
use TEXT_IO, C_LIB, CALENDAR;

procedure TACTICAL3 is

 OOD: OBJECT_ID := AUV_OOD.class_object;
 DR : OBJECT_ID := DATA_RECORDER.class_object;
 MM : OBJECT_ID := MISSION_MODEL.class_object;
 SR : OBJECT_ID := SENSORY_RECEIVER.class_object;
 WM : OBJECT_ID := WORLD_MODEL.class_object;


 CODE : BOOLEAN;
 LOOP_COUNT : INTEGER := 0;
 BEGIN_TIME, END_TIME : TIME;
 TOT_SECS : DURATION;
 goal : INTEGER;
 GOAL_FLAG : INTEGER;
 STATUS_NUM : INTEGER;

 package FLOAT_INOUT is new FLOAT_IO(FLOAT);
 use FLOAT_INOUT;
 package INTEGER_INOUT is new INTEGER_IO(INTEGER);
 use INTEGER_INOUT;
 package FIXED_INOUT is new FIXED_IO(DURATION);
 use FIXED_INOUT;

 procedure PRINT_HEADER is
 begin
 NEW_LINE;
 PUT_LINE("Welcome to the startup program!");
 NEW_LINE;
 end PRINT_HEADER;

 procedure PRINT_TRAILER is
 begin
 PUT_LINE("Simulation complete!");
 NEW_LINE;
 end PRINT_TRAILER;

 procedure INSTANTIATE_ALLES is
```

```
begin
PUT_LINE("Instantiations beginning from top level...");
send(DR, CREATE, new_data_recorder => DR);
send(MM, CREATE, new_mission_model => MM);
send(SR, CREATE, new_sensory_receiver => SR);
send(WM, CREATE, new_world_model => WM);
send(OOD, CREATE, new_auv_ood => OOD);
send(OOD, NAV_OOD_BACKLINK, OOD => OOD);
send(OOD, ENGR_OOD_BACKLINK, OOD => OOD);
send(OOD, WEAP_OOD_BACKLINK, OOD => OOD);
PUT_LINE("Everything's created!");
NEW_LINE;
end INSTANTIATE_ALLES;

procedure DELETE_ALLES is
begin
PUT_LINE("Deletions beginning from top level...");
send(OOD, DELETE);
send(DR, DELETE);
send(MM, DELETE);
send(SR, DELETE);
send(WM, DELETE);
PUT_LINE("Everything's gone!");
NEW_LINE;
end DELETE_ALLES;

procedure LINK_OBJECTS is
begin
PUT_LINE("Revealing handles of sub-objects...");
send(OOD, LINKUP_MM, pointer => MM);
send(OOD, LINKUP_SR, pointer => SR);
send(SR, LINKUP_DR, DR => DR);
end LINK_OBJECTS;

procedure OPEN_NETWORK is
begin
PUT("Connecting tactical to execution now...");
start_comms;
NEW_LINE;
PUT("Starting tactical server, listening for strategic level...");
start_iris_s;
PUT_LINE("Done!");
end OPEN_NETWORK;

procedure READY_VEHICLE_FOR_LAUNCH is
begin
PUT_LINE("Downloading mission parameters...");
PUT_LINE("Obstacles first: x, y, and depth.");
send(OOD, DOWNLOAD_OBSTACLES);
PUT_LINE("Then the init state: x, y, depth, and heading.");
send(OOD, DOWNLOAD_INITIAL_STATE);
```

```
PUT_LINE("Then the waypoints: speed, x, y, and depth.");
send(OOD, DOWNLOAD_WAYPOINTS);
PUT_LINE("Finally, the goal point: x_final and y_final.");
send(OOD, DOWNLOAD_FINAL_GOAL);
PUT_LINE("State loaded and systems checks completed.");
        GOAL_FLAG := 1; -- Should query OOD for success
end READY_VEHICLE_FOR_LAUNCH;

procedure SELECT_FIRST_WAYPOINT is
begin
PUT_LINE("Selecting first waypoint, generating first ");
PUT_LINE(" heading command, and sending it.");
send(OOD, SEL_1ST_WP);
        GOAL_FLAG := 1;
        NEW_LINE;
        PUT_LINE("Mission initiated...");
        NEW_LINE;
end SELECT_FIRST_WAYPOINT;

procedure ALERT_USER is
begin
        NEW_LINE;
        PUT_LINE("Failure occurred during mission download and/or ");
        PUT_LINE("pre-mission checks. Please correct fault before ");
        PUT_LINE("proceeding.");
        NEW_LINE;
        GOAL_FLAG := 1;
end ALERT_USER;

procedure IN_TRANSIT_P is
begin
GOAL_FLAG := 0;
end IN_TRANSIT_P;

procedure TRANSIT_DONE_P is
begin
GOAL_FLAG := 1;
end TRANSIT_DONE_P;

procedure IN_SEARCH_P is
begin
GOAL_FLAG := 0;
end IN_SEARCH_P;

procedure SEARCH_DONE_P is
begin
GOAL_FLAG := 1;
end SEARCH_DONE_P;

procedure IN_TASK_P is
begin
```

```
GOAL_FLAG := 0;
end IN_TASK_P;

procedure TASK_DONE_P is
begin
GOAL_FLAG := 1;
end TASK_DONE_P;

procedure IN_RETURN_P is
begin
GOAL_FLAG := 1;
end IN_RETURN_P;

procedure RETURN_DONE_P is
begin
send(OOD, REACH_GOAL, ans => CODE);
if CODE then
GOAL_FLAG := 1;
NEW_LINE;
PUT_LINE("**********Goal Reached**********");
NEW_LINE;
else
GOAL_FLAG := 0;
end if;
end RETURN_DONE_P;

procedure WAIT_FOR_RECOVERY is
begin
GOAL_FLAG := 1;
end WAIT_FOR_RECOVERY;

procedure SURFACE is
begin
   send(OOD, SURFACE);
GOAL_FLAG := 1;
end SURFACE;

procedure DO_SEARCH_PATTERN is
begin
GOAL_FLAG := 1;
end DO_SEARCH_PATTERN;

procedure HOMING is
begin
GOAL_FLAG := 1;
end HOMING;

procedure DROP_PACKAGE is
begin
GOAL_FLAG := 1;
end DROP_PACKAGE;
```

```
procedure GET_GPS_FIX is
begin
GOAL_FLAG := 1;
end GET_GPS_FIX;

procedure GET_NEXT_WAYPOINT is
begin
send(OOD, GET_NEXT_WP);
GOAL_FLAG := 1;
end GET_NEXT_WAYPOINT;

procedure SEND_WAYPOINTS_AND_MODES is
begin
send(OOD, EXECUTE_PLAN);
GOAL_FLAG := 1;
end SEND_WAYPOINTS_AND_MODES;

procedure REACH_WAYPOINT is
begin
send(OOD, REACH_WAYPOINT, ans => CODE);
if CODE then
GOAL_FLAG := 1;
PUT_LINE("Waypoint obtained! Selecting next waypoint...");
get_time;
else
GOAL_FLAG := 0;
end if;
send(OOD, GET_VEH_POSTURE);
send(OOD, PLAN);
end REACH_WAYPOINT;

procedure GPS_NEEDED is
begin
GOAL_FLAG := 0;
end GPS_NEEDED;

procedure UNKNOWN_OBSTACLE_P is
begin
GOAL_FLAG := 0;
end UNKNOWN_OBSTACLE_P;

procedure LOG_NEW_OBSTACLE is
begin
GOAL_FLAG := 1;
end LOG_NEW_OBSTACLE;

procedure LOITER is
begin
GOAL_FLAG := 1;
end LOITER;
```

```
procedure START_LOCAL_REPLANNER is
begin
GOAL_FLAG := 1;
end START_LOCAL_REPLANNER;

procedure START_GLOBAL_REPLANNER is
begin
GOAL_FLAG := 1;
end START_GLOBAL_REPLANNER;

procedure POWER_GONE_P is
 begin
 send(OOD, POWER_CHECK, ans => STATUS_NUM);
 GOAL_FLAG := 0;
 end POWER_GONE_P;

 procedure COMPUTER_SYSTEM_PROB_P is
 begin
 GOAL_FLAG := 0;
 end COMPUTER_SYSTEM_PROB_P;

 procedure PROPULSION_SYSTEM_PROB_P is
 begin
 GOAL_FLAG := 0;
 end PROPULSION_SYSTEM_PROB_P;

 procedure STEERING_SYSTEM_PROB_P is
 begin
 GOAL_FLAG := 0;
 end STEERING_SYSTEM_PROB_P;

 procedure DIVING_SYSTEM_PROB_P is
 begin
 GOAL_FLAG := 0;
 end DIVING_SYSTEM_PROB_P;

 procedure BOUYANCY_SYSTEM_PROB_P is
 begin
 GOAL_FLAG := 0;
 end BOUYANCY_SYSTEM_PROB_P;

 procedure THRUSTER_SYSTEM_PROB_P is
 begin
 GOAL_FLAG := 0;
 end THRUSTER_SYSTEM_PROB_P;

 procedure LEAK_TEST_P is
 begin
 GOAL_FLAG := 0;
 end LEAK_TEST_P;
```

```
procedure PAYLOAD_PROB_P is
begin
GOAL_FLAG := 0;
end PAYLOAD_PROB_P;


procedure CLOSE_NETWORK is
begin
PUT_LINE("Closing tactical/execution network...");
stop_comms;
        PUT_LINE("Closing down tactical level server...");
        stop_iris_s;
end CLOSE_NETWORK;

procedure PRINT_PROFILE is
begin

TOT_SECS := SECONDS(END_TIME) - SECONDS(BEGIN_TIME);
NEW_LINE;
PUT("For the mission, the total execution time was: ");
PUT(TOT_SECS);
NEW_LINE;
PUT("and the total control cycles taken was: ");
PUT(LOOP_COUNT);
NEW_LINE;
PUT("Yielding a control rate of ");
PUT(FLOAT(LOOP_COUNT)/FLOAT(TOT_SECS));
PUT(" commands per second.");
NEW_LINE;

end PRINT_PROFILE;


begin
PRINT_HEADER;
INSTANTIATE_ALLES;
LINK_OBJECTS;
OPEN_NETWORK;
goal := get_from_strat_i;

loop
-- PUT_LINE("Primitive goal received. Initiating behavior...");
LOOP_COUNT := LOOP_COUNT + 1;
case goal is
        when 10 => READY_VEHICLE_FOR_LAUNCH;
         put_iris_i_s(GOAL_FLAG); -- Download mission, etc.
        when 11 => SELECT_FIRST_WAYPOINT;
         BEGIN_TIME := CLOCK; -- Start mission timer
get_time;
        put_iris_i_s(GOAL_FLAG);
```

```
when 12 => ALERT_USER;
 put_iris_i_s(GOAL_FLAG);
 exit;
when 13 => IN_TRANSIT_P;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, in_transit_p)
when 14 => TRANSIT_DONE_P;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, transit_done_p)
when 15 => IN_SEARCH_P;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, in_search_p)
when 16 => SEARCH_DONE_P;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, search_done_p)
when 17 => IN_TASK_P;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, in_task_p)
when 18 => TASK_DONE_P;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, task_done_p)
when 19 => IN_RETURN_P;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, in_return_p)
when 20 => RETURN_DONE_P;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, return_done_p)
when 21 => WAIT_FOR_RECOVERY;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, wait_for_recovery)
 exit;
when 22 => SURFACE;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, surface)
when 23 => DO_SEARCH_PATTERN;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, do_search_pattern)
when 24 => HOMING;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, homing)
when 25 => DROP_PACKAGE;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, drop_package)
when 26 => GET_GPS_FIX;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, get_gps_fix)
when 27 => GET_NEXT_WAYPOINT;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, get_next_waypoint)
when 28 => SEND_WAYPOINTS_AND_MODES;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, send_waypoints_and_modes)
when 29 => REACH_WAYPOINT;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, reach_waypoint)
when 30 => GPS_NEEDED;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, gps_needed)
when 31 => UNKNOWN_OBSTACLE_P;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, unknown_obstacle_p)
when 32 => LOG_NEW_OBSTACLE;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, log_new_obstacle)
when 33 => LOITER;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, loiter)
when 34 => START_LOCAL_REPLANNER;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, start_local_replanner)
when 35 => START_GLOBAL_REPLANNER;
 put_iris_i_s(GOAL_FLAG); -- send(OOD, start_global_replanner)
when 36 => POWER_GONE_P;
```

```
            put_iris_i_s(GOAL_FLAG); -- send(OOD, power_gone_p)
           when 37 => COMPUTER_SYSTEM_PROB_P;
            put_iris_i_s(GOAL_FLAG); -- send(OOD, computer_system_prob_p)
           when 38 => PROPULSION_SYSTEM_PROB_P;
            put_iris_i_s(GOAL_FLAG); -- send(OOD, propulsion_system_prob_p)
           when 39 => STEERING_SYSTEM_PROB_P;
            put_iris_i_s(GOAL_FLAG); -- send(OOD, steering_system_prob_p)
           when 40 => DIVING_SYSTEM_PROB_P;
            put_iris_i_s(GOAL_FLAG); -- send(OOD, diving_system_prob_p)
           when 41 => BOUYANCY_SYSTEM_PROB_P;
            put_iris_i_s(GOAL_FLAG); -- send(OOD, bouyancy_system_prob_p)
           when 42 => THRUSTER_SYSTEM_PROB_P;
            put_iris_i_s(GOAL_FLAG); -- send(OOD, thruster_system_prob_p)
           when 43 => LEAK_TEST_P;
            put_iris_i_s(GOAL_FLAG); -- send(OOD, leak_test_p)
           when 44 => PAYLOAD_PROB_P;
            put_iris_i_s(GOAL_FLAG); -- send(OOD, payload_prob_p)
           when others => PUT("Unexpected goal value received: ");
            PUT(goal);
            NEW_LINE;
    end case;
    goal := get_from_strat_i;
    end loop;

    END_TIME := CLOCK;
    get_time;
    CLOSE_NETWORK;
    DELETE_ALLES;
    PRINT_TRAILER;
    PRINT_PROFILE;

    end TACTICAL3;
```

APPENDIX C. INTER-LEVEL COMMUNICATIONS SOFTWARE

1. PROLOG-TO-ADA, PROLOG SIDE

```
/* This is the Prolog-to-C "glue" software
 Date created: 30 Nov 92
 Date Revised: 9 Dec 92


To be run on any workstation. NOTE: to obtain an object file xxx.o, invoke the compiler
as follows:

 cc -c xxx.c

where xxx is the file name. The -lm flag must be placed after the name of the source file if
linkage to the math.h library is desired. */

#include <stdio.h>
#include <time.h>

ready_vehicle_for_launch_p() /* Perform systems status check and download
 mission parameters. Integers passed to Tactical level represent primitive goals. */
{
        int result;

        printf("%s\n", "Entered the function ready_vehicle_for_launch_p");
        start_link();
        put_iris_i(10);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

select_first_waypoint()
{
        int result;

        printf("%s\n", "Entered the function select_first_waypoint");
        put_iris_i(11);
        printf("\n");
        get_iris_i(&result);
        return (result);
}
```

```
alert_user()
{
        int result;

        printf("%s\n", "Entered the function alert_user");
        put_iris_i(12);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

in_transit_p() {
        int result;

        printf("%s\n", "Entered the function in_transit_p");
        put_iris_i(13);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

transit_done_p() {
        int result;

        printf("%s\n", "Entered the function transit_done_p");
        put_iris_i(14);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

in_search_p() {
        int result;

        printf("%s\n", "Entered the function in_search_p");
        put_iris_i(15);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

search_done_p() {
        int result;

        printf("%s\n", "Entered the function search_done_p");
        put_iris_i(16);
        printf("\n");
        get_iris_i(&result);
        return (result);
}
```

```
in_task_p() {
        int result;

        printf("%s\n", "Entered the function in_task_p");
        put_iris_i(17);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

task_done_p() {
        int result;

        printf("%s\n", "Entered the function task_done_p");
        put_iris_i(18);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

in_return_p() {
        int result;

        printf("%s\n", "Entered the function in_return_p");
        put_iris_i(19);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

return_done_p() {
        int result;

        printf("%s\n", "Entered the function return_done_p");
        put_iris_i(20);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

wait_for_recovery() {
        int result;

        printf("%s\n", "Entered the function wait_for_recovery");
        put_iris_i(21);
        printf("\n");
        get_iris_i(&result);
        /* Delay may be needed to insure server has time to shut down */
        stop_link();
        return (result);
}
```

```
surface() {
        int result;

        printf("%s\n", "Entered the function surface");
        put_iris_i(22);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

do_search_pattern() {
        int result;

        printf("%s\n", "Entered the function do_search_pattern");
        put_iris_i(23);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

homing() {
        int result;

        printf("%s\n", "Entered the function homing");
        put_iris_i(24);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

drop_package() {
        int result;

        printf("%s\n", "Entered the function drop_package");
        put_iris_i(25);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

get_gps_fix() {
        int result;

        printf("%s\n", "Entered the function get_gps_fix");
        put_iris_i(26);
        printf("\n");
        get_iris_i(&result);
        return (result);
}
```

```
get_next_waypoint() {
        int result;

        printf("%s\n", "***Waypoint attained***");
        printf("%s\n", "Entered the function get_next_waypoint");
        put_iris_i(27);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

send_setpoints_and_modes() {
        int result;

        printf("%s\n", "Entered the function send_setpoints_and_modes");
        put_iris_i(28);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

reach_waypoint_p() {
        int result;

        printf("%s\n", "Entered the function reach_waypoint_p");
        put_iris_i(29);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

gps_needed_p() {
        int result;

        printf("%s\n", "Entered the function gps_needed_p");
        put_iris_i(30);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

unknown_obstacle_p() {
        int result;

        printf("%s\n", "Entered the function unknown_obstacle_p");
        put_iris_i(31);
        printf("\n");
        get_iris_i(&result);
        return (result);
}
```

```
log_new_obstacle() {
        int result;

        printf("%s\n", "Entered the function log_new_obstacle");
        put_iris_i(32);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

loiter() {
        int result;

        printf("%s\n", "Entered the function loiter");
        put_iris_i(33);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

start_local_replanner() {
        int result;

        printf("%s\n", "Entered the function start_local_replanner");
        put_iris_i(34);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

start_global_replanner() {
        int result;

        printf("%s\n", "Entered the function start_global_replanner");
        put_iris_i(35);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

power_gone_p() {
        int result;

        printf("%s\n", "Entered the function power_gone_p");
        put_iris_i(36);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

computer_system_inop_p() {
```

```
        int result;

        printf("%s\n", "Entered the function computer_system_inop_p");
        put_iris_i(37);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

propulsion_system_p() {
        int result;

        printf("%s\n", "Entered the function propulsion_system_p");
        put_iris_i(38);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

steering_system_inop_p() {
        int result;

        printf("%s\n", "Entered the function steering_system_inop_p");
        put_iris_i(39);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

diving_system_p() {
        int result;

        printf("%s\n", "Entered the function diving_system_p");
        put_iris_i(40);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

bouyancy_system_p() {
        int result;

        printf("%s\n", "Entered the function bouyancy_system_p");
        put_iris_i(41);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

thruster_system_p() {
        int result;
```

```
        printf("%s\n", "Entered the function thruster_system_p");
        put_iris_i(42);
        printf("\n");
        get_iris_i(&result);
        return (result);
}

leak_test_p() {
        int result;

        printf("%s\n", "Entered the function leak_test_p");
        put_iris_i(43);
        printf("\n");
        get_iris_i(&result);
        return (result);
}


payload_prob_p() {
        int result;

        printf("%s\n", "Entered the function payload_prob_p");
        put_iris_i(44);
        printf("\n");
        get_iris_i(&result);
        return (result);
}


get_time()
{
        time_t now;
        now = time(NULL);
        printf("%s%s\n", " The time is now = ", ctime(&now));
}


/************************ communications software ********************/


start_link()
{
        start_iris("gemini"); /* hardwire tactical level host (server) specified here */
        return(99);
}

stop_link()
{
        stop_iris();
        return(100);
```

```
}

/**$$*********************************************************************/

/*************************************************************************
;
; Filename.....: client.c
;
; This program creates a socket and returns the socket.
; The following function is available.
;
; 1. Created by Sehung Kwak 10/10/90
;
; usage : connect_to_server(remote_server_host_name, port_number)
; ; returns a socket
;
;*************************************************************************/
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

/*
 Return socket
*/

connect_to_server(remote_server_host_name, port_number)

char *remote_server_host_name;
int port_number;


{
        int sock;
        struct sockaddr_in server;
        struct hostent *hp, *gethostbyname();
        char buf[1024];

        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
        perror("opening stream socket");
        exit(1);
        }

        server.sin_family = AF_INET;
        hp = gethostbyname(remote_server_host_name);
        if (hp == 0) {
        fprintf(stderr, "%s: unknown host\n", remote_server_host_name);
        exit(2);
        }
```

```
        bcopy((char *)hp->h_addr, (char *)&server.sin_addr, hp->h_length);
        server.sin_port = htons(port_number);

        if (connect(sock, (struct sockaddr *)&server, sizeof server) < 0) {
        perror("connecting stream socket");
        exit(1);
}       else
        return(sock);
        }

/
*************************************************************************
;
; Filename.....: comm_glue1c.c
;
; This takes care of socket stream communication interface.
; Following functions are available.
; A Sun running this program should be client.
;
; 1. Created by Sehung Kwak 10/10/90
; 2. Modified for C language interface, Sehung Kwak 1/27/92
;
; usage : open_stream_c (host_name port_number) ; open stream
; write_string_c (string) ; write string
; read_string_c (string, size) ; read string
; write_char_c (length_one_string) ; write character
; force_out_c () ; output write buffer
; read_char_c () ; read character
; close_stream_c () ; close stream
;
; This file needs client.c.
;*************************************************************************/

#define TRUE 1
#define FALSE 0
#define BUFSIZE 1024

static int sock;
static char sbuf[BUFSIZE];
static int sptr = 0;

open_stream_c (host_name, port_num)
char host_name[];
long port_num;
{
        sock = connect_to_server(host_name,(int) port_num);

        bzero(sbuf, sizeof sbuf); /* initialize send buffer */
        sptr = 0; /* initialize send buffer pointer */

        if (sock >= 0)
```

```
        return(TRUE);
        else
        return(FALSE);
}

write_string_c (ps)
char *ps;
{
        if (write(sock, ps, strlen(ps)) < 0) {
        perror("Writing on stream socket");
        return(FALSE);
        } else
        return(TRUE);
}

force_out_c ()
{
        if (write_string_c(sbuf) == TRUE) {
        bzero(sbuf, sizeof sbuf);
        sptr = 0;
        return(TRUE);
        } else
        return(FALSE);
}


write_char_c (ps)
char *ps;
{
        sbuf[sptr++]=*ps;
        if (sptr == BUFSIZE)
        force_out_c();
}

read_string_c (str, size)
char str[];
int size;
{
        if (read(sock, str, size) < 0) {
        perror("Reading stream socket");
        return(FALSE);
        } else
        return(TRUE);
}

char *read_char_c ()
{
        char onestr[4];
        bzero(onestr, sizeof onestr);
        if (read(sock, onestr, 1) < 0) {
        perror("Reading stream socket");
```

```
        return('\0');
        } else
        return(onestr);
}

close_stream_c()
{
        if (close(sock) < 0)
        return(FALSE);
        else
        return(TRUE);
}


/
*************************************************************************
;
;
; Filename .....: convert.c
;
; This program converts a number (float or integer) or a string
; to a communiction format and also records received data to
; a number(float or integer) or a string.
;
; Communication Format
;
; TXXXXDDDDDDDD....
;
; T : type (one of I, R, C. i.e., integer, float, string)
; XXXX : data size in byte, filled with leading zeros
; DDD... : actual data (sequence of ASCII characters)
;
; example: I0003123
;          ^^^^^^^
; TXXXXDDD
;
;
; 1. Created by Sehung Kwak 1/27/92
;
; usage: float_to_data(float, data) ; float --> data
; integer_to_data(integer, data) ; integer --> data
; string_to_data(string,data) ; string --> data
; data_to_float(data,float*) ; data --> float
; data_to_integer(data,integer*) ; data --> integer
; data_to_string(data,string) ; data --> string
;
;
;*************************************************************************/

#include <stdio.h>;
```

```
#define MAX_FLOAT_SIZE 30
#define MAX_INTEGER_SIZE 30


void float_to_data(x, data)
float x;
char data[];
{
        char x_str[MAX_FLOAT_SIZE];

        sprintf(x_str, "%f", x);
        sprintf(data, "R%04d%s", strlen(x_str), x_str);
}


void integer_to_data(x, data)
int x;
char data[];
{
        char x_str[MAX_INTEGER_SIZE];

        sprintf(x_str, "%d", x);
        sprintf(data, "I%04d%s", strlen(x_str), x_str);
}


void string_to_data(str, data)
char str[], data[];
{
        sprintf(data, "C%04d%s", strlen(str), str);
}


void data_to_float(data, px)
char data[];
float* px;
{
        char type;
        int size;

        sscanf(data,"%1c%4d%f", &type, &size, px);
}

void data_to_integer(data, px)
char data[];
int* px;
{
        char type;
        int size;

        sscanf(data,"%1c%4d%d", &type, &size, px);
```

```
}


void data_to_string(data, str)
char data[], str[];
{
        char type;
        int size;

        sscanf(data, "%1c%4d%s", &type, &size, str);
}

/
*************************************************************************
;
; Filename.....: sun_iris_comms1c.c
;
; 1. Created by Sehung Kwak 1/27/92
; Use one socket for communication.
;
; *local-port* 1053
;
;
;
; usage : start_iris ("<target server ID>") ; make connection
; put_iris_s (string) ; send string data
; put_iris_f (float) ; send floating point number
; put_iris_i (integer) ; send integer number
; get_iris_s (string) ; get string data
; get_iris_f (&float) ; get float data (ptr)
; get_iris_i (&integer) ; get integer data (ptr)
; stop_iris() ; close communication
;
;
; This file needs convert.c and comm_glue1c.c
;************************************************************************/

#include <stdio.h>;

#define TRUE 1
#define FALSE 0
#define LOCAL_PORT 1051
#define STR_BUFSIZE 1024
#define NUM_BUFSIZE 30

start_iris(host_name)
char host_name[];
{
        if (open_stream_c(host_name,LOCAL_PORT))
        return(TRUE);
        else
```

```
        return(FALSE);
}


stop_iris()
{
        return(close_stream_c());
}


put_iris_s(str)
char str[];
{
        char buf[STR_BUFSIZE];

        string_to_data(str,buf);
        write_string_c(buf);
        force_out_c();
        return(TRUE);
}


put_iris_f(num_f)
float num_f;
{
        char buf[NUM_BUFSIZE];

        float_to_data(num_f, buf);
        write_string_c(buf);
        force_out_c();
        return(TRUE);
}

put_iris_i(num_i)
int num_i;
{
        char buf[NUM_BUFSIZE];

        bzero(buf, sizeof buf);
        integer_to_data(num_i, buf);
        printf("Int to data is %s\n", buf);

        write_string_c(buf);
        return(TRUE);
}


get_iris_s(str)
char *str;
{
        char buf[STR_BUFSIZE];
```

```
        read_string_c(buf,STR_BUFSIZE);
        data_to_string(buf, str);
        return(TRUE);
}


get_iris_f(pf)
float *pf;
{
        char buf[NUM_BUFSIZE];

        read_string_c(buf,NUM_BUFSIZE);
        data_to_float(buf, pf);
        return(TRUE);
}

get_iris_i(pi)
int *pi;
{
        char buf[NUM_BUFSIZE];

        bzero(buf, sizeof buf);
        read_string_c(buf,NUM_BUFSIZE);
        data_to_integer(buf, pi);
        return(TRUE);
}

/
*****************************************************************************
*****************************************************************************
*****************************************************************************/
```

2. PROLOG-TO-ADA, ADA SIDE

```
package C_LIB is
--
-- CLIENT comms, supporting Tactical<->Execution level
--
  procedure start_comms; -- make connection to E-level
  procedure put_float (X : FLOAT); -- send float to E-level
  function get_float return FLOAT; -- receive float from E-level
  procedure put_mode (X : INTEGER); -- send mode to E-level
  procedure stop_comms; -- close connection to E-level
--
-- SERVER comms, supporting Strategic<->Tactical level
--
  procedure start_iris_s; -- establish term as server
  procedure put_iris_i_s (X : INTEGER); -- send int to S-level
  function get_from_strat_i return INTEGER;-- receive int from S-level
  procedure stop_iris_s;
--
-- System clock access function. Better than Ada's
--
  procedure get_time;

private

  pragma INTERFACE(C, start_comms);
  pragma INTERFACE(C, put_float);
  pragma INTERFACE(C, get_float);
  pragma INTERFACE(C, stop_comms);
  pragma INTERFACE(C, put_mode);
  pragma INTERFACE(C, start_iris_s);
  pragma INTERFACE(C, put_iris_i_s);
  pragma INTERFACE(C, get_from_strat_i);
  pragma INTERFACE(C, stop_iris_s);
  pragma INTERFACE(C, get_time);


  pragma LINK_WITH("network_sw.o"); -- lump all above files together

end C_LIB;


/* Network comms for the Tactical level. The Tactical level must act
 as a server for the Strategic level and a client for the Execution
 level. Code with a filename ending in "s" is server code; those
 files ending in "c" are client code.
```

```
 Date: 12 Jan 93
*/

#include <sys/types.h>          /* server.c, client.c */
#include <sys/socket.h>         /* server.c, client.c */
#include <netinet/in.h>         /* server.c, client.c */
#include <netdb.h>              /* server.c, client.c */
#include <stdio.h>              /* server.c, convert.c, sun_iris_comms1s.c, client.c,
                                   sun_iris_comms1c.c */
#include <string.h>             /* sun_iris_comms1c.c */

#include <time.h>

#define TRUE 1                  /* comm_glue_1s.c, sun_iris_comms1s.c,
                                   comm_glue_1c.c, sun_iris_comms1c.c */
#define FALSE 0                 /* comm_glue_1s.c, sun_iris_comms1s.c,
                                   comm_glue_1c.c, sun_iris_comms1c.c */
#define BUFSIZE 1024            /* comm_glue_1s.c, comm_glue_1c.c */
#define MAX_FLOAT_SIZE 30       /* convert.c */
#define MAX_INTEGER_SIZE 30     /* convert.c */
#define STR_BUFSIZE 1024        /* sun_iris_comms1s.c, sun_iris_comms1c.c */
#define NUM_BUFSIZE 30          /* sun_iris_comms1s.c, sun_iris_comms1c.c */
#define GET_DATA "get_data"     /* sun_iris_comms1s.c, sun_iris_comms1c.c */
#define ST_LOCAL_PORT 1051      /* sun_iris_comms1s.c */
#define TE_LOCAL_PORT 1052      /* sun_iris_comms1c.c */

static int sock_s, sock_c;      /* comm_glue_1s.c, comm_glue_1c.c */
static char sbuf_s[BUFSIZE];
static char sbuf_c[BUFSIZE];    /* comm_glue_1s.c, comm_glue_1c.c */
static int sptr_s = 0;          /* comm_glue_1s.c, comm_glue_1c.c */
static int sptr_c = 0;


/*****************************************************************************
;
; Filename.....: client.c
;
; This program creates a socket and returns the socket.
; The following function is available.
;
; 1. Created by Sehung Kwak 10/10/90
;
;
; usage : connect_to_server(remote_server_host_name, port_number)
; ; returns a socket
;
;
;*****************************************************************************/

/*
 Return socket
```

```
*/

connect_to_server(remote_server_host_name, port_number)

char *remote_server_host_name;
int port_number;


{
        int sock;
        struct sockaddr_in server;
        struct hostent *hp, *gethostbyname();
        char buf[1024];

        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
        perror("opening stream socket");
        exit(1);
        }

        server.sin_family = AF_INET;
        hp = gethostbyname(remote_server_host_name);
        if (hp == 0) {
        fprintf(stderr, "%s: unknown host\n", remote_server_host_name);
        exit(2);
        }
        bcopy((char *)hp->h_addr, (char *)&server.sin_addr, hp->h_length);
        server.sin_port = htons(port_number);

        if (connect(sock,
         (struct sockaddr *)&server, sizeof server) < 0) {
        perror("connecting stream socket");
        exit(1);
        } else
        return(sock);
}


/*****************************************************************************
;
; Filename.....: server.c
;
; This program creates a socket and returns the socket.
; The following function is available.
;
; 1. Created by Sehung Kwak 12/3/91
;
;
; usage : start_server(port_number)
; ; returns a socket
;
```

```
;
;*****************************************************************/


/*
 Returns socket
*/
int start_server(port_number)
int port_number; /* server port number */

{
        int sock;
        struct sockaddr_in server;
        int msgsock;

        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
        perror("opening stream socket");
        exit(1);
        }

        server.sin_family = AF_INET;
        server.sin_addr.s_addr = INADDR_ANY;
        server.sin_port = port_number;
        if (bind(sock, (struct sockaddr *)&server,
         sizeof server) < 0) {
        perror("binding stream socket");
        exit(1);
 }

        listen(sock,5);

        msgsock = accept(sock, (struct sockaddr *)0, (int *)0);
        if (msgsock == -1) {
        perror("accept");
        exit(2);
        } else
        return(msgsock);
}


/
*****************************************************************
;
; Filename.....: comm_glue1c.c
;
; This takes care of socket stream communication interface.
; Following functions are available.
; A Sun running this program should be client.
;
; 1. Created by Sehung Kwak 10/10/90
```

```
; 2. Modified for C language interface, Sehung Kwak 1/27/92
;
;
; usage : open_stream_c (host_name port_number) ; open stream
; write_string_c (string) ; write string
; read_string_c (string, size) ; read string
; write_char_c (length_one_string) ; write character
; force_out_c () ; output write buffer
; read_char_c () ; read character
; close_stream_c () ; close stream
;
;
; This file needs client.c.
;*************************************************************************/


open_stream_c (host_name, port_num)
char host_name[];
long port_num;
{
        sock_c = connect_to_server(host_name,(int) port_num);

        bzero(sbuf_c, sizeof sbuf_c); /* initialize send buffer */
        sptr_c = 0; /* initialize send buffer pointer */

        if (sock_c >= 0)
        return(TRUE);
        else
        return(FALSE);
}

write_string_c (ps)
char *ps;
{
        if (write(sock_c, ps, strlen(ps)) < 0) {
        perror("Writing on stream socket");
        return(FALSE);
        } else
        return(TRUE);
}


force_out_c ()
{
        if (write_string_c(sbuf_c) == TRUE) {
        bzero(sbuf_c, sizeof sbuf_c);
        sptr_c = 0;
        return(TRUE);
        } else
        return(FALSE);
}
```

```
write_char_c (ps)
char *ps;
{
        sbuf_c[sptr_c++]=*ps;
        if (sptr_c == BUFSIZE)
        force_out_c();

}


read_string_c (str, size)
char str[];
int size;
{
        if (read(sock_c, str, size) < 0) {
        perror("Reading stream socket");
        return(FALSE);
        } else
        return(TRUE);
}

char *read_char_c ()
{
        char onestr[4];
        bzero(onestr, sizeof onestr);
        if (read(sock_c, onestr, 1) < 0) {
        perror("Reading stream socket");
        return('\0');
        } else
        return(onestr);
}


close_stream_c()
{
        if (close(sock_c) < 0)
        return(FALSE);
        else
        return(TRUE);
}


/
*****************************************************************************
;
; Filename.....: comm_glue1s.c
;
; This takes care of socket stream communication interface.
; Following functions are available.
```

```
; A Sun running this program should be server.
;
; 1. Created by Sehung Kwak 12/3/92
;
;
; usage : open_stream_s (port_number) ; open write stream
; write_string_s (string) ; write string
; read_string_s (string, size) ; read string
; write_char_s (length_one_string) ; write character
; force_out_s () ; output write buffer
; read_char_s () ; read character
; close_stream_s () ; close stream
;
;
; This file needs server.c.
;*************************************************************************/


open_stream_s (port_num)
long port_num;
{
        sock_s = start_server((int) port_num);

        bzero(sbuf_s, sizeof sbuf_s); /* initialize send buffer */
        sptr_s = 0; /* initialize send buffer pointer */

        if (sock_s >= 0)
        return(TRUE);
        else
        return(FALSE);
}


write_string_s (ps)
char *ps;
{
        if (write(sock_s, ps, strlen(ps)) < 0) {
        perror("Writing on stream socket");
        return(FALSE);
        } else
        return(TRUE);
}


force_out_s ()
{
        if (write_string_s(sbuf_s) == TRUE) {
        bzero(sbuf_s, sizeof sbuf_s);
        sptr_s = 0;
        return(TRUE);
        } else
```

```
        return(FALSE);
}


write_char_s (ps)
char *ps;
{

        sbuf_s[sptr_s++]= *ps;
        if (sptr_s == BUFSIZE)
        force_out_s();

}


read_string_s (str, size)
char str[];
int size;
{
        if (read(sock_s, str, size) < 0) {
        perror("Reading stream socket");
        return(FALSE);
        } else
        return(TRUE);
}

char *read_char_s ()
{
        char onestr[4];
        bzero(onestr, sizeof onestr);
        if (read(sock_s, onestr, 1) < 0) {
        perror("Reading stream socket");
        return('\0');
        } else
        return(onestr);
}


close_stream_s()
{
        if (close(sock_s) < 0)
        return(FALSE);
        else
        return(TRUE);
}


/*****************************************************************************
;
;
; Filename .....: convert.c
```

```
;
; This program converts a number (float or integer) or a string
; to a communiction format and also records received data to
; a number(float or integer) or a string.
;
; Communication Format
;
; TXXXXDDDDDDDD....
;
; T : type (one of I, R, C. i.e., integer, float, string)
; XXXX : data size in byte, filled with leading zeros
; DDD... : actual data (sequence of ASCII characters)
;
; example: I0003123
; ^^^^^^^
; TXXXXDDD
;
;
; 1. Created by Sehung Kwak 1/27/92
;
; usage: float_to_data(float, data) ; float --> data
; integer_to_data(integer, data) ; integer --> data
; string_to_data(string,data) ; string --> data
; data_to_float(data,float*) ; data --> float
; data_to_integer(data,integer*) ; data --> integer
; data_to_string(data,string) ; data --> string
;
;
;*************************************************************************/


void float_to_data(x, data)
float x;
char data[];
{
        char x_str[MAX_FLOAT_SIZE];

        sprintf(x_str, "%f", x);
        sprintf(data, "R%04d%s", strlen(x_str), x_str);
}


void integer_to_data(x, data)
int x;
char data[];
{
        char x_str[MAX_INTEGER_SIZE];

        sprintf(x_str, "%d", x);
        sprintf(data, "I%04d%s", strlen(x_str), x_str);
}
```

```
void string_to_data(str, data)
char str[], data[];
{
        sprintf(data, "C%04d%s", strlen(str), str);
}


void data_to_float(data, px)
char data[];
float* px;
{
        char type;
        int size;

        sscanf(data,"%1c%4d%f", &type, &size, px);
}

void data_to_integer(data, px)
char data[];
int* px;
{
        char type;
        int size;

        sscanf(data,"%1c%4d%d", &type, &size, px);
}


void data_to_string(data, str)
char data[], str[];
{
        char type;
        int size;

        sscanf(data, "%1c%4d%s", &type, &size, str);
}


/
**************************************************************************
;
; Filename.....: sun_iris_comms1c.c
;
; 1. Created by Sehung Kwak 1/27/92
; Use one socket for communication.
;
; *tac-to-ex comms-port* 1052
;
;
```

```
;
; usage : start_iris ("<target_server>") ; make connection
; put_iris_s (string) ; send string data
; put_iris_f (float) ; send floating point number
; put_iris_i (integer) ; send integer number
; get_iris_s (string) ; get string data
; get_iris_f (&float) ; get float data (ptr)
; get_iris_i (&integer) ; get integer data (ptr)
; stop_iris() ; close communication
;
;
; This file needs convert.c and comm_glue1c.c
;*************************************************************************/


start_iris(host_name)
char host_name[];
{
        if (open_stream_c(host_name,TE_LOCAL_PORT))
        return(TRUE);
        else
        return(FALSE);
}


stop_iris()
{
        return(close_stream_c());
}


int get_ack_c()
{
        char buf[STR_BUFSIZE];
        char str[10];

        bzero(buf, sizeof buf);
        read_string_c(buf,STR_BUFSIZE);
        data_to_string(buf,str);
        if (!(strcmp(str,GET_DATA)))
        return 1;
        else
        return 0;
}

int send_ack_c()
{
        char buf[STR_BUFSIZE];

        string_to_data(GET_DATA,buf);
```

```
        write_string_c(buf);
}


put_iris_s(str)
char str[];
{
        char buf[STR_BUFSIZE];

        string_to_data(str,buf);
        write_string_c(buf);
        if (!(get_ack_c())) {
        printf("Error: No acknowledgement \n");
        }
        return(TRUE);
}


put_iris_f(num_f)
float num_f;
{
        char buf[NUM_BUFSIZE];

        float_to_data(num_f, buf);
        write_string_c(buf);
        if (!(get_ack_c())) {
        printf("Error: No acknowledgement \n");
        }
        return(TRUE);
}

put_iris_i(num_i)
int num_i;
{
        char buf[NUM_BUFSIZE];

        integer_to_data(num_i, buf);
        write_string_c(buf);
        if (!(get_ack_c())) {
        printf("Error: No acknowledgement \n");
        }
        return(TRUE);
}


get_iris_s(str)
char *str;
{
        char buf[STR_BUFSIZE];

        read_string_c(buf,STR_BUFSIZE);
```

```
        data_to_string(buf, str);
        send_ack_c();
        return(TRUE);
 }


get_iris_f(pf)
float *pf;
{
        char buf[NUM_BUFSIZE];

        read_string_c(buf,NUM_BUFSIZE);
        data_to_float(buf, pf);
        send_ack_c();
        return(TRUE);
}

get_iris_i(pi)
int *pi;
{
        char buf[NUM_BUFSIZE];

        read_string_c(buf,NUM_BUFSIZE);
        data_to_integer(buf, pi);
        send_ack_c();
        return(TRUE);
}


/
*****************************************************************************
;
; Filename.....: sun_iris_comms1s.c
;
; 1. Created by Sehung Kwak 12/3/92
; Use one socket for communication.
;
; *local-port* 1053
;
;
;
; usage : start_iris_s () ; make connection
; put_iris_s_s (string) ; send string data
; put_iris_f_s (float) ; send floating point number
; put_iris_i_s (integer) ; send integer number
; get_iris_s_s (string) ; get string data
; get_iris_f_s (&float) ; get float data (ptr)
; get_iris_i_s (&integer) ; get integer data (ptr)
; stop_iris_s() ; close communication
;
;
```

```
; This file needs server.c and comm_glue1s.c
;*****************************************************************/



start_iris_s()
{
        if (open_stream_s(ST_LOCAL_PORT))
        return(TRUE);
        else
        return(FALSE);
}


stop_iris_s()
{
        return(close_stream_s());
}

put_iris_s_s(str)
char str[];
{
        char buf[STR_BUFSIZE];

        string_to_data(str,buf);
        write_string_s(buf);
        force_out_s();
        return(TRUE);
}


put_iris_f_s(num_f)
float num_f;
{
        char buf[NUM_BUFSIZE];

        float_to_data(num_f, buf);
        write_string_s(buf);
        force_out_s();
        return(TRUE);
}

put_iris_i_s(num_i)
int num_i;
{
        char buf[NUM_BUFSIZE];

        bzero(buf, sizeof buf);
        integer_to_data(num_i, buf);
        write_string_s(buf);
        force_out_s();
```

```
        return(TRUE);
}


get_iris_s_s(str)
char *str;
{
        char buf[STR_BUFSIZE];

        read_string_s(buf,STR_BUFSIZE);
        data_to_string(buf, str);
        return(TRUE);
}

get_iris_f_s(pf)
float *pf;
{
        char buf[NUM_BUFSIZE];

        read_string_s(buf,NUM_BUFSIZE);
        data_to_float(buf, pf);
        return(TRUE);
}

get_iris_i_s(pi)
int *pi;
{
        char buf[NUM_BUFSIZE];

        bzero(buf, sizeof buf);
        read_string_s(buf,NUM_BUFSIZE);
        data_to_integer(buf, pi);
        return(TRUE);
}

get_from_strat_i()
{
        int x;
        get_iris_i_s(&x);
        return(x);
}


/
*****************************************************************************
;
; The following are additional "access" routines used to link
; sun_iris_comms1c.c routines to Ada code without resorting to
; passing of pointers.
;
; 1. Created by Ron Byrnes 11/19/92; revised 11/25/92
```

```
;
; usage : start_comms() ; avoids need to pass strings between Ada and C
; put_float(cmd) ; send floating point number
; get_float () ; extracts value from ptr to avoid passing addresses between Ada and C
; stop_comms() ; maintains consistent labelling
;
;*****************************************************************/

void start_comms()
{
        printf("Starting comms with iris.\n\n");
        start_iris("iris1");
}

void stop_comms()
{
        printf("Stopping comms with iris.\n\n");
        stop_iris();
}

void put_float(cmd)
float cmd;
{
        put_iris_f(cmd);
}

float get_float()
{
        float temp;
        get_iris_f(&temp);
        return(temp);
}

void put_mode(cmd)
int cmd;
{
        put_iris_s("TRANSIT"); /* At this time, sim has only one mode */
}

get_time()
{
        time_t now;
        now = time(NULL);
        printf("%s%s\n", "The time is now = ", ctime(&now));
}

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

3. ADA-TO-C, C SIDE

```
/
*****************************************************************************
;
; Filename.....: comm_glue1s.c
;
; This takes care of socket stream communication interface.
; Following functions are available.
; An Iris running this program should be server.
;
; 1. Created by Sehung Kwak 12/3/92
;
;
; usage : open_stream_s (port_number) ; open write stream
; write_string_s (string) ; write string
; read_string_s (string, size) ; read string
; write_char_s (length_one_string) ; write character
; force_out_s () ; output write buffer
; read_char_s () ; read character
; close_stream_s () ; close stream
;
;
; This file needs server.c.
;*****************************************************************************/

#define TRUE 1
#define FALSE 0
#define BUFSIZE 1024

static int sock;
static char sbuf[BUFSIZE];
static int sptr = 0;

open_stream_s (port_num)
long port_num;
{
        sock = start_server((int) port_num);

        bzero(sbuf, sizeof sbuf); /* initialize send buffer */
        sptr = 0; /* initialize send buffer pointer */

        if (sock >= 0)
        return(TRUE);
        else
        return(FALSE);
```

```
}


write_string_s (ps)
char *ps;
{
        if (write(sock, ps, strlen(ps)) < 0) {
        perror("Writing on stream socket");
        return(FALSE);
        } else
        return(TRUE);
}


force_out_s ()
{
        if (write_string_s(sbuf) == TRUE) {
        bzero(sbuf, sizeof sbuf);
        sptr = 0;
        return(TRUE);
        } else
        return(FALSE);
}


write_char_s (ps)
char *ps;
{

        sbuf[sptr++]= *ps;
        if (sptr == BUFSIZE)
        force_out_s();

}


read_string_s (str, size)
char str[];
int size;
{
        if (read(sock, str, size) < 0) {
        perror("Reading stream socket");
        return(FALSE);
        } else
        return(TRUE);
}

char *read_char_s ()
{
        char onestr[4];
        bzero(onestr, sizeof onestr);
```

```
        if (read(sock, onestr, 1) < 0) {
        perror("Reading stream socket");
        return('\0');
        } else
        return(onestr);
}


close_stream_s()
{
        if (close(sock) < 0)
        return(FALSE);
        else
        return(TRUE);
}


/
*************************************************************************
;
;
; Filename .....: convert.c
;
; This program converts a number (float or integer) or a string
; to a communiction format and also records received data to
; a number(float or integer) or a string.
;
; Communication Format
;
; TXXXXDDDDDDDD....
;
; T : type (one of I, R, C. i.e., integer, float, string)
; XXXX : data size in byte, filled with leading zeros
; DDD... : actual data (sequence of ASCII characters)
;
; example: I0003123
; ^^^^^^^
; TXXXXDDD
;
;
; 1. Created by Sehung Kwak 1/27/92
;
; usage: float_to_data(float, data) ; float --> data
; integer_to_data(integer, data) ; integer --> data
; string_to_data(string,data) ; string --> data
; data_to_float(data,float*) ; data --> float
; data_to_integer(data,integer*) ; data --> integer
; data_to_string(data,string) ; data --> string
;
;
;*************************************************************************/
```

```
#include <stdio.h>;

#define MAX_FLOAT_SIZE 30
#define MAX_INTEGER_SIZE 30


void float_to_data(x, data)
float x;
char data[];
{
        char x_str[MAX_FLOAT_SIZE];

        sprintf(x_str, "%f", x);
        sprintf(data, "R%04d%s", strlen(x_str), x_str);
}


void integer_to_data(x, data)
int x;
char data[];
{
        char x_str[MAX_INTEGER_SIZE];

        sprintf(x_str, "%d", x);
        sprintf(data, "I%04d%s", strlen(x_str), x_str);
}


void string_to_data(str, data)
char str[], data[];
{
        sprintf(data, "C%04d%s", strlen(str), str);
}


void data_to_float(data, px)
char data[];
float* px;
{
        char type;
        int size;

        sscanf(data,"%1c%4d%f", &type, &size, px);
}

void data_to_integer(data, px)
char data[];
int* px;
{
        char type;
```

```
        int size;

        sscanf(data,"%1c%4d%d", &type, &size, px);
}


void data_to_string(data, str)
char data[], str[];
{
        char type;
        int size;

        sscanf(data, "%1c%4d%s", &type, &size, str);
}



/*************************************************************************
;
; Filename.....: server.c
;
; This program creates a socket and returns the socket.
; The following function is available.
;
; 1. Created by Sehung Kwak 12/3/91
;
;
; usage : start_server(port_number)
; ; returns a socket
;
;
;*************************************************************************/
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>


/*
 Returns socket
*/
int start_server(port_number)

int port_number; /* server port number */

{
        int sock;
        struct sockaddr_in server;
        int msgsock;
```

```
        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
        perror("opening stream socket");
        exit(1);
        }

        server.sin_family = AF_INET;
        server.sin_addr.s_addr = INADDR_ANY;
        server.sin_port = port_number;
        if (bind(sock, (struct sockaddr *)&server,
         sizeof server) < 0) {
        perror("binding stream socket");
        exit(1);
        }

        listen(sock,5);

        msgsock = accept(sock,
         (struct sockaddr *)0, (int *)0);
        if (msgsock == -1) {
        perror("accept");
        exit(2);
        } else
        return(msgsock);
}


/
*************************************************************************
;
; Filename.....: sun_iris_comms1s.c
;
; 1. Created by Sehung Kwak 12/3/92
; Use one socket for communication.
;
; *local-port* 1052
;
;
;
; usage : start_iris_s () ; make connection
; put_iris_s_s (string) ; send string data
; put_iris_f_s (float) ; send floating point number
; put_iris_i_s (integer) ; send integer number
; get_iris_s_s (string) ; get string data
; get_iris_f_s (&float) ; get float data (ptr)
; get_iris_i_s (&integer) ; get integer data (ptr)
; stop_iris_s() ; close communication
;
;
; This file needs server.c and comm_glue1s.c
;*************************************************************************/
```

```
#include <stdio.h>
#include <string.h>

#define TRUE 1
#define FALSE 0
#define LOCAL_PORT 1052
#define STR_BUFSIZE 1024
#define NUM_BUFSIZE 30
#define GET_DATA "get_data"

start_iris_s()
{
        printf("Listening for Tactical level...\n\n");
        if (open_stream_s(LOCAL_PORT))
        return(TRUE);
        else
        return(FALSE);
}


stop_iris_s()
{
        return(close_stream_s());
}


int get_ack_s()
{
        char buf[STR_BUFSIZE];
        char str[10];

        bzero(buf, sizeof buf);
        read_string_s(buf,STR_BUFSIZE);
        data_to_string(buf,str);
        if (!(strcmp(str,GET_DATA)))
        return 1;
        else
        return 0;
}

int send_ack_s()
{
        char buf[STR_BUFSIZE];

        string_to_data(GET_DATA,buf);
        write_string_s(buf);
}
```

```
put_iris_s_s(str)
char str[];
{
        char buf[STR_BUFSIZE];

        string_to_data(str,buf);
        write_string_s(buf);
        if (!(get_ack_s())) {
        printf("Error: No acknowledgement \n");
 }
 return(TRUE);
 }


put_iris_f_s(num_f)
float num_f;
{
        char buf[NUM_BUFSIZE];

        float_to_data(num_f, buf);
        write_string_s(buf);
        if (!(get_ack_s())) {
        printf("Error: No acknowledgement \n");
        }
        return(TRUE);
}

put_iris_i_s(num_i)
int num_i;
{
        char buf[NUM_BUFSIZE];

        integer_to_data(num_i, buf);
        write_string_s(buf);
        if (!(get_ack_s())) {
        printf("Error: No acknowledgement \n");
        }
        return(TRUE);
}


get_iris_s_s(str)
char *str;
{
        char buf[STR_BUFSIZE];

        read_string_s(buf,STR_BUFSIZE);
        data_to_string(buf, str);
        send_ack_s();
        return(TRUE);
}
```

```
get_iris_f_s(pf)
float *pf;
{
        char buf[NUM_BUFSIZE];

        read_string_s(buf,NUM_BUFSIZE);
        data_to_float(buf, pf);
        send_ack_s();
        return(TRUE);
}

get_iris_i_s(pi)
int *pi;
{
        char buf[NUM_BUFSIZE];

        read_string_s(buf,NUM_BUFSIZE);
        data_to_integer(buf, pi);
        send_ack_s();
        return(TRUE);
}


get_from_strat_i()
{
        int x;
        get_iris_i_s(&x);
        return(x);
}
```

APPENDIX D. GLOSSARY OF TERMINOLOGY FOR AUTONOMOUS VEHICLE SOFTWARE ARCHITECTURES

Autonomous Vehicle: a self-contained mobile robot with the capacity to sense a dynamic and unstructured environment, plan an intelligent response to that input, and act in a way that is compatible with the accomplishment of a mission without human intervention.

Behavior: an algorithm designed specifically to generate the numeric input required by a feedback control system which will, in turn, produce a change in the underlying physical plant consistent with the desired primitive goal which activated the behavior. The term *task* is a frequently used synonym.

Doctrine: the part of the RBM Strategic level containing the logic required to solve problems not unique to the particular mission at hand. Doctrine is in general vehicle dependent.

Execution Level: the lowest level of the Rational Behavior Model containing all the software required to meet hard real-time deadlines while ensuring basic vehicle stability and safety.

Goal: a result toward which effort, provided by an external entity, is directed. The problem, prior to decomposition, is the *root goal.* The set of goals not further decomposed are called *primitive goals*, and all other subgoals are called *intermediate goals.*

Goal Tree: a graphical representation of AND/OR goal decomposition, with the root node representing the root goal, the leaf nodes representing the primitive goals, all other nodes representing intermediate goals subject to further decomposition, and the connecting arcs

representing the logical relationship between subgoals and the goal from which they were decomposed.

Layer of Control: the realization of a single behavior or competence of the underlying system.

Level of Control: a distinct set of computational entities characterized by a shared conceptual abstraction, of which temporal, spatial, and command hierarchies are the most common.

Mission Model: a database comprising the information necessary to uniquely identify each segment or phase of the mission and the terminating conditions for each phase.

Mission Specification: the part of the RBM Strategic level containing the rule set embodying mission specific knowledge.

Problem Decomposition: the successive simplification of a root goal, resulting in intermediate and primitive goals which may be placed in a tree structure to represent the orderly search for a problem solution.

Rational Behavior Model-Backward (RBM-B): a form of RBM characterized by a backward-chaining inference mechanism at the strategic level employing goal decomposition and an AND/OR goal tree as the basis for its search.

Rational Behavior Model-Forward (RBM-F): a form of RBM characterized by a forward-chaining inference mechanism at the Strategic level employing a state transition diagram to organize the sequence of allowable state transitions.

Software Architecture: a structural plan encompassing the conceptual design and organization of software. The architecture includes, but is not limited to, a description of the abstraction mechanisms, division of responsibility, and specification of external interfaces through which the software entities communicate with each other and the external world. Each software entity, also called a *module*, is an independent conceptual unit within a software system.

State Transition Diagram with Path Priorities: models the time-dependent behavior of a system containing state transition conditions which may not be mutually exclusive. Conflicts in determining the next state are resolved through the use of pre-assigned, numerical path priorities.

World Model: the set of data reflecting the characteristics of the environment external to the vehicle and upon which the control software bases its decisions.

LIST OF REFERENCES

[1] Weisbin, C. R., et al., "Autonomous Mobile Robot Navigation and Learning, *Computer*, pp. 29-35, June 1989.

[2] Gevarter, W. B., *Intelligent Machines: An Introductory Perspective of Artificial Intelligence and Robotics*, Prentice-Hall, 1985.

[3] Brogan, W. L., *Modern Control Theory*, 2d ed., Prentice-Hall 1985.

[4] Hutchinson, S. A. and Kak, A. C., "Spar: A Planner That Satisfies Operational and Geometric Goals in Uncertain Environments", *AI Magazine*, Vol. 11, No. 1, Spring 1990, pp. 30-61.

[5] *Autonomous Mobile Robots*, I. J. Cox and G. T. Wilfong, eds., Springer-Verlag, 1990.

[6] *Record of the Architectural Approaches Working Group at the First International IEEE Workshop on Architectures for Real-Time Intelligent Control Systems (ARTICS)*, Arlington, Virginia, January 1991.

[7] Harmon, S. Y., "The Ground Surveillance Robot (GSR): An Autonomous Vehicle Designed to Transit Unknown Terrain", *IEEE Journal of Robotics and Automation*, Vol. RA-3, No. 3, June 1987.

[8] Brooks, F. P., *The Mythical Man-Month*, Addison-Wesley, 1975.

[9] *Webster's New World Dictionary*, 2d college ed., Simon and Schuster, 1982.

[10] Simmons, R. G. and R. J. Firby, "Robot Architectures", *Tutorial SA5 presented at the Tenth National Conference on Artificial Intelligence (AAAI-92)*, San Jose, California, July 12, 1992.

[11] *Proceedings of the 1992 International Conference on Robotics and Automation*, May 12-14, 1992, Nice, France.

[12] Fu, K. S., R. C. Gonzalez, and C. S. G. Lee, *Robotics: Control, Sensing, Vision, and Intelligence*, McGraw-Hill, 1987.

[13] Davis, D. "Control of MBARI ROV Cameras and Tools Over a Network", *Proceedings of the First IARP Workshop on Mobile Robots for Subsea Environments*, Monterey, California, October 23-26, 1990, pp. 137-142.

[14] Yoerger, D. R., "Precise Control of Underwater Robots: Why and How", *Proceedings of the First IARP Workshop on Mobile Robots for Subsea Environments*, Monterey, California, October 23-26, 1990, pp. 113-117.

[15] Shaker, S. M. and A. R. Wise, *War Without Men*, Pergamin-Brassey's, 1988.

[16] Nilsson, N. J., "A Mobile Automaton: An Application of Artificial Intelligence Techniques", *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 509-520, Washington, D. C., May 7-9, 1969.

[17] Moravec, H. P., "The Stanford Cart and the CMU Rover", *Proceedings of the IEEE*, Vol. 71, No. 7, pp. 872-884, July 1983.

[18] Meystel, A., *Autonomous Mobile Robots*, World Scientific, 1991.

[19] *Autonomous Mobile Robots: Control, Planning, and Architecture*, S. S. Iyengar and A. Elfes, eds., IEEE Computer Society, 1991.

[20] Kumar, V. R. and K. J. Waldron, "A Review of Research on Walking Machines", *Robotics Review 1*, MIT Press, pp. 243-266, 1989.

[21] McGhee, R. B., "Vehicular Legged Locomotion", in *Advances in Automation and Robotics*, G. N. Saridis, ed., pp. 259-284, Jai Press, 1985.

[22] Kwak, S. H., *A Computer Simulation Study of a Free Gait Motion Coordination Algorithm for Rough-Terrain Locomotion by a Hexapod Walking Machine*, Ph.D. Dissertation, The Ohio State University, Columbus, Ohio, 1986.

[23] Kwak, S. H. and R. B. McGhee, "Rule-Based Motion Coordination for a Hexapod Walking Machine", *Advanced Robotics*, Vol. 4, No. 3, pp. 263-282, 1990.

[24] McCarthy, J. and P. J. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence", *Machine Intelligence*, Vol. 4, pp. 463-502, 1969.

[25] Hayes, P. J., "The Frame Problem and Related Problems in Artificial Intelligence", in *Artificial and Human Thinking*, A. Elithorn and D. Jones, eds., Jossey-Bass, 1973.

[26] Prior, A. N., *Past, Present, and Future*, Clarendon Press, 1967.

[27] Nishimura, H., "Descriptively Complete Process Logic", *Acta Informatica*, Vol. 14, No. 4, pp. 359-369, October 1980.

[28] Fikes, R. E. and N. J. Nilsson, "STRIPS: a New Approach to the Application of Theorem Proving to Problem Solving", *Artificial Intelligence*, Vol. 2, No. 3-4, pp. 189-208, Winter 1971.

[29] Georgeff, M. P., "Planning", *Annual Review of Computer Science, Vol. 2*, pp. 359-400, Annual Reviews, 1987.

[30] Albus, J. S., *Brains, Behavior, and Robotics*, McGraw-Hill, 1981.

[31] Brooks, R. A., "A Robust Layered Control System for a Mobile Robot", *IEEE Journal of Robotics and Automation*, Vol. RA-2, No. 1, pp. 14-23, 1986.

[32] Brooks, R. A., "A Robot That Walks: Emergent Behaviors From a Carefully Evolved Network", *Neural Computation*, Vol. 1, No. 2, pp. 253-262, 1989.

[33] Everett, H. R., et al., *Modeling the Environment of a Mobile Security Robot*, Naval Ocean Systems Center Technical Document 1835, San Diego, California, June 1990.

[34] Nilsson, N. J., *Triangle Tables: A Proposal for a Robot Programming Language*, Technical Note 347, SRI AI Center, Menlo Park, California, 1985.

[35] Crowley, J. L., "Navigation For An Intelligent Mobile Robot", *IEEE Journal of Robotics and Automation*, Vol. RA-1, pp.31-41, 1985.

[36] *An Introduction to Intelligent and Autonomous Control*, P. J. Antsaklis and K. M. Passino, eds., Kluwer Academic Publishers, 1993.

[37] Saridis, G., "Intelligent Robotic Control", *IEEE Transactions on Automatic Control*, Vol. AC-28, No. 5, pp. 547-556, May 1983.

[38] Alami R., R. Chatila, and P. Freedman, "Task Level Teleprogramming for Intervention Robots", *Proceedings of the First IARP Workshop on Mobile Robots for Subsea Environments*, Monterey, California, October 23-26, 1990, pp. 119-136.

[39] Blidberg, D. R., et al., "The EAVE AUV Program at the Marine Systems Engineering Laboratory", *Proceedings of the First IARP Workshop on Mobile Robots for Subsea Environments*, pp. 33-42, Monterey, California, October 23-26, 1990.

[40] Nilsson, N. J., *Shakey the Robot*, Technical Note 323, SRI AI Center, Menlo Park, California, April 1984.

[41] Kan, E. and E. Austin, "The JPL Telerobot Teleoperation System", *International Journal of Robotics and Automation*, Vol. 5, No. 1, pp. 27-31, 1990.

[42] Herman M. and J. Albus, "Overview of the Multiple Autonomous Underwater Vehicles (MAUV) Project", *Proceedings of the 1988 IEEE International Conference on Robotics and Automation*, Philadelphia, Pennsylvania, April 24-29, 1988, pp. 618-620.

[43] Albus, J. S., H. G. McCain, and R. Lumia, *NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)*, NIST Technical Note 1235, 1989.

[44] Bihari, T. E., T. M. Walliser, and M. R. Patterson, "Controlling The Adaptive Suspension Vehicle", *Computer*, Vol. 22, No. 6, pp. 59-65, June 1989.

[45] Payton, D. W. and T. E. Bihari, "Intelligent Real-Time Control Of Robotic Vehicles", *Communications of the ACM*, Vol. 34, No. 8, pp. 49-63, August 1991.

[46] Barr, A., P. Cohen, and E. A. Feigenbaum., *The Handbook of Artificial Intelligence*, Vols. 1-4, William Kaufmann, 1981-86.

[47] Lamb, D. A., *Software Engineering: Planning for Change*, Prentice-Hall, 1988.

[48] Brooks, R. A., "Intelligence Without Representation", *Artificial Intelligence*, Vol. 47, No. 1-3, pp. 139-159, January 1991.

[49] Freedman, D. H., "Invasion of the Insect Robots", *Discover*, Vol. 12, No. 3, pp. 42-50, March 1991.

[50] Zheng, X., "Layered Control of a Practical AUV", *Proceedings of the 1992 Symposium on Autonomous Underwater Vehicle Technology (AUV '92)*, pp. 142-147, Washington, D. C., June 2-3, 1992.

[51] Payton, D. W., "An Architecture for Reflexive Autonomous Vehicle Control", *Proceedings of the 1986 IEEE International Conference on Robotics and Automation*, pp. 1838-1845, San Francisco, California, April 7-10, 1986.

[52] Payton, D. W., J. K. Rosenblatt, and D. M. Keirsey, "Plan Guided Reaction", *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 20, No. 6, pp. 1370-1382, November/December 1990.

[53] Rosenblatt, J. K. and D. W. Payton, "A Fine-Grained Alternative To The Subsumption Architecture For Mobile Robot Control", *Proceedings of the International Joint Conference on Neural Networks*, June 1989, Washington, D.C., pp. 317-323.

[54] Mataric, M. J., "Integration of Representation into Goal-Driven Behavior-Based Robots", *IEEE Transactions on Robotics and Automation*, Vol. 8, No. 3, pp. 304-312, June 1992.

[55] Zheng, X., E. Jackson, and M. Kao, "Object-Oriented Software Architecture For Mission-Configurable Robots", *Proceedings of the First IARP Workshop on Mobile Robots for Subsea Environments*, October 23-26, 1990, Monterey, California, pp. 63-73.

[56] Bellingham, J. G. and T. R. Consi, "State Configured Layered Control", *Proceedings of the First IARP Workshop on Mobile Robots for Subsea Environments*, pp. 75-80, Monterey, California, October 23-26, 1990.

[57] Daily, M., et al., "Autonomous Cross-Country Navigation With The ALV", *Proceedings of the 1988 IEEE International Conference on Robotics and Automation*, pp. 718-726, Philadelphia, Pennsylvania, April 24-29, 1988.

[58] Coste-Manière, E., B. Espiau, and E. Rutten, "A Task-Level Robot Programming Language and its Reactive Execution", *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, pp. 2751-2756, Nice, France, May 12-14, 1992.

[59] Byrnes, R. B., et al., "An Experimental Comparison Of Hierarchical And Subsumption Software Architectures For Control Of An Autonomous Underwater Vehicle", *Proceedings of the 1992 Symposium on Autonomous Underwater Vehicle Technology (AUV '92)*, pp. 135-141, Washington, D. C., June 2-3, 1992.

[60] Day, D. S., *Achieving Flexibility for Autonomous Agents in Dynamic Environments*, Ph. D. Dissertation, University of Massachusetts, May 1991.

[61] Rodseth, O. J., "Object-Oriented Software System for AUV Control", *Proceedings of the First IARP Workshop on Mobile Robots for Subsea Environments*, Monterey, California, October 23-26, 1990, pp. 15-24.

[62] Gat, E., *Reliable Goal-Directed Reactive Control of Autonomous Mobile robots*, Ph. D. Dissertation, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, April 1991.

[63] Manber, U., *Introduction to Algorithms: A Creative Approach*, Addison-Wesley, 1989.

[64] Boehm, B. W., "A Spiral Model of Software Development and Enhancement", *Computer*, Vol. 21, No. 5, pp. 61-72, May 1988.

[65] Bares, J., et al., "Ambler: An Autonomous Rover for Planetary Exploration", *Computer*, Vol. 22, No. 6, pp. 18-26, June 1989.

[66] *Blackboard Systems*, R. Engelmore and T. Morgan, eds., Addison-Wesley, 1988.

[67] Pappas, G., et al., "The DARPA/Navy Unmanned Undersea Vehicle Program", *Unmanned Systems*, Vol. 9, No. 2, pp. 24-30, Spring 1991.

[68] *Autonomous Control Logic Concept Final Report*, Naval Coastal Systems Center/Lockheed AI Center, December 1990.

[69] Turk, M. A., et al., "VITS—A Vision System for Autonomous Land Vehicle Navigation", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 10, No. 3, pp. 342-361, May 1988.

[70] Olin, K. E. and D. Y. Tseng, "Autonomous Cross-Country Navigation: An Integrated Perception and Planning System", *IEEE Expert*, Vol.6, No. 4, pp. 16-30, August 1991.

[71] Brutzman, D. P. and M. A. Compton, "AUV Research at the Naval Postgraduate School", *Sea Technology*, pp. 35-40, December 1991.

[72] Healey, A. J., et al., "Research on Autonomous Underwater Vehicles at the Naval Postgraduate School", *Naval Research Reviews*, Vol. 44, No. 1, pp. 43-51, 1992.

[73] Brutzman, D. P., Y. Kanayama, and M. J. Zyda, "Integrated Simulation for Rapid Development of Autonomous Underwater Vehicles", *Proceedings of the 1992 Symposium on Underwater Vehicle Technology (AUV '92)*, pp. 3-10, Washington, D. C., June 2-3, 1992.

[74] MacLennan, B. J., *Principles of Programming Languages*, 2d ed., Holt, Rinehart and Winston, 1987.

[75] Sethi, R., *Programming Languages: Concepts and Constructs*, Addison-Wesley, 1990.

[76] Backus, J., "The History of FORTRAN I, II, and III", *ACM SIGPLAN Notices*, Vol. 13, No. 8, pp. 165-180, August 1978.

[77] Jensen, R. W., "Structured Programming", *Computer*, Vol. 14, No. 3, pp. 31-48, March 1981.

[78] Wilson, L. B. and R. G. Clark, *Comparative Programming Languages: A Conceptual Approach*, Addison-Wesley, 1988.

[79] Winston, P. H. and B. K. P. Horn, *LISP*, 3d ed., Addison-Wesley, 1989.

[80] McCarthy, J., "History of LISP", *ACM SIGPLAN Notices*, Vol. 13, No. 8, pp. 217-223, August 1978.

[81] Savitch, W. J., *Pascal, An Introduction to the Art and Science of Programming*, 2d ed., Benjamin/Cummings, 1987.

[82] Gabriel, R. P., J. L. White, and D. G. Bobrow, "CLOS: Integrating Object-Oriented and Functional Programming", *Communications of the ACM*, Vol. 34, No. 9, pp. 29-38, September 1991.

[83] Robinson, J. A., "Logic and Logic Programming", *Communications of the ACM*, Vol. 35, No. 3, pp. 40-65, March 1992.

[84] Kowalski, R. A., "Algorithms = Logic + Control", *Communications of the ACM*, Vol. 22, No. 7, pp. 424-436, July 1979.

[85] Sterling, L. and E. Shapiro, *The Art of Prolog*, MIT Press, 1986.

[86] Snyder, A., "Encapsulation and Inheritance in Object-Oriented Programming Systems, Languages and Applications", *ACM SIGPLAN Notices*, Vol. 21, No.11, pp. 38-45, November 1986.

[87] Nelson, M. L., *An Introduction to Object-Oriented Programming*, Technical Report NPS52-90-024, Naval Postgraduate School, April 1990.

[88] Booch, G., *Object Oriented Design with Applications*, Benjamin/Cummings, 1991.

[89] Wegner, P. "Dimensions of Object-Based Language Design", special issue of *SIGPLAN Notices*, Vol. 22, No. 12, pp. 168-182, December 1987.

[90] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, 1986.

[91] Moon, D. A., "The COMMON LISP Object-Oriented Programming Language Standard", in *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. H. Lochovsky, eds., pp. 49-78, ACM Press/Addison-Wesley, 1989.

[92] *Classic-Ada User's Manual*, Software Productivity Solutions, 1992.

[93] Goldberg A. and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[94] Meyer, B., *Object-Oriented Software Construction*, Prentice-Hall, 1988.

[95] Nelson, M. L., "An Object-Oriented Tower of Babel", *OOPS Messenger*, Vol. 2, No. 3, pp. 3-11, July 1991.

[96] Ben-Ari, M., *Principles of Concurrent Programming*, Prentice-Hall International, 1982.

[97] Burns, A., *Concurrent Programming in Ada*, Cambridge University Press, 1985.

[98] Nelson, M. L., "Concurrent and Distributed Object-Oriented Languages: Promises and Pitfalls", *Keynote Address, NATO Defence Research Group of the RSG.1 Workshop on Object-Oriented Modeling of Distributed Systems*, Quebec, Canada, May 12-15, 1992.

[99] Beizer, B., *Software Testing Techniques*, 2d ed., Von Nostrand Reinhold, 1990.

[100] Deitel, H. M., *Operating Systems*, 2d ed., Addison-Wesley, 1990.

[101] *Tutorial: Software Testing and Validation Techniques*, 2d ed., E. Miller and W. E. Howden, eds., IEEE Computer Society Press, 1981.

[102] Carver, R. H. and K. C. Tai, "Replay and Testing for Concurrent Programs", *IEEE Software*, Vol. 8, No. 2, pp. 66-74, May 1991.

[103] Quinn, M. J., *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, 1987.

[104] Schwan, K. and H. Zhou, "Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads", *IEEE Transactions on Software Engineering*, Vol. 18, No. 8, pp. 736-748, August 1992.

[105] Burns, A. and A. Wellings, *Real-Time Systems and Their Programming Languages*, Addison-Wesley, 1990.

[106] Badr, S. M., et al., *Real-Time Systems*, Technical Report NPSCS-92-004, Naval Postgraduate School, Monterey, California, February 1992.

[107] Fox, G., et al., *Solving Problems on Concurrent Processors, Vol. 1*, Prentice-Hall, 1988.

[108] Lin, K.-J. and E. J. Burke, "Coming to Grips with Real-Time Realities", *IEEE Software*, Vol. 9, No. 5, pp. 12-15, September 1992.

[109] Stankovic, J. A. and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Systems", *IEEE Software*, Vol. 8, No. 3, pp. 62-72, May 1991.

[110] Kenny, K. B. and K.-J. Lin, "Building Flexible Real-Time Systems Using the Flex Language", *Computer*, Vol. 24, No. 5, pp. 70-78, May 1991.

[111] Berry, G. and L. Cosserat, "The ESTEREL Synchronous Programming Language and its Mathematical Semantics", *Lecture Notes in Computer Science 197: Seminar on Concurrency*, pp. 389-448, Springer-Verlag, 1985.

[112] *Ada 9X Project Report: Ada 9X Requirements*, Office of the Under Secretary of Defense for Acquisition, Washington, D. C., December 1990.

[113] Nelson, M. L., *Object-Oriented Real-Time Computing*, Technical Report NPSCS-92-010, Naval Postgraduate School, Monterey, California, August 1992.

[114] Luqi, "Computer-Aided Prototyping for Command and Control Systems Using CAPS", *IEEE Software*, Vol. 9, No. 1, pp. 56-67, January 1992.

[115] Jackson, P., *Introduction to Expert Systems*, 2d ed., Addison-Wesley, 1990.

[116] Webber, B. L. and N. J. Nilsson, Preface to *Readings in Artificial Intelligence*, Morgan Kaufmann, 1981.

[117] Kowalski, R. A., *Logic for Problem Solving*, North-Holland, 1979.

[118] Davis, R., and King, J., "An Overview of Production Systems", *Machine Intelligence*, Vol. 8, John Wiley, 1977.

[119] Findler, N. V., *Associative Networks*, Academic Press, 1979.

[120] Manna, Z., and Waldinger, R., *The Logical Basis for Computer Programming, Volume 1: Deductive Reasoning*, Addison-Wesley, 1985.

[121] Roth, C. H., *Fundamentals of Logic Design*, West, 1985.

[122] Nolt, J. and D. Rohatyn, *Schaum's Outline of Theory and Problems of Logic*, McGraw-Hill, 1988, pp. 39-73, 116-164.

[123] Rowe, N. C., *Artificial Intelligence Through Prolog*, Prentice Hall, 1988.

[124] Brooke, T., "The Art of Production Systems", *AI Expert*, Vol. 7, No. 1, pp. 31-35, January 1992.

[125] Robinson, J. A., "A Machine-Oriented Logic Based on the Resolution Principle", *Journal of the Association for Computing Machinery*, vol. 12, pp. 23-41, 1965.

[126] White, L. J. and M. L. Nelson, *Resolution and Unification Using Resolve*, Technical Report NPS52-90-021, Naval Postgraduate School, Monterey, California, March 1990.

[127] Nilsson, N. J., *Principles of Artificial Intelligence*, Tioga, 1980.

[128] Simmons, A. B. and S. G. Chappell, "Artificial Intelligence—Definition and Practice", *IEEE Journal of Oceanic Engineering*, Vol. 13, No. 2, pp. 14-42, April 1988.

[129] Minsky, M., "Logical Versus Analogical or Symbolic Versus Connectionist or Neat Versus Scruffy", *AI Magazine*, Vol. 12, No. 2, pp. 34-51, Summer 1991.

[130] Post, E. L., "Formal Reductions of the General Combinatorial Decision Problem", *American Journal of Mathematics*, vol. 65, pp. 197-268, 1943.

[131] Shortliffe, E. H., *MYCIN: Computer-based Medical Consultations*, Elsevier, 1976.

[132] Newell, A., "The Knowledge Level", *Artificial Intelligence*, Vol. 18, No. 1, pp. 87-127, January 1982.

[133] Winston, P. H., *Artificial Intelligence*, 3d ed., Addison-Wesley, 1992.

[134] Giarratano, J. C., *CLIPS User's Guide*, Ver. 5.0, Software Technology Branch, Johnson Space Center, NASA, January 1991.

[135] Kiper, J. D., "Structural Testing of Rule-Based Expert Systems", *ACM Transactions on Software Engineering and Methodology*, Vol. 1, No. 2, pp. 168-187, April 1992.

[136] Yourdon, E., *Modern Structured Analysis*, Yourdon Press, 1989.

[137] Sanderson, A. C., L. S. Homem de Mello, and H. Zhang, "Assembly Sequence Planning", *AI Magazine*, pp. 62-81, Spring 1990.

[138] Hart, P. E., Nilsson, N. J., and Raphael, B., "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", *IEEE Transactions on Systems, Science, and Cybernetics*, Vol. SSC-4, No. 2, pp. 100-107, July 1968.

[139] Nagao, M., *Knowledge and Inference*, Academic Press, 1990.

[140] Kwak, S. H., T. Scholz, and R. B. Byrnes, *The State Transition Diagram With Path Priorities and Its Applicability to the Translation Between Backward and Forward Implementations of the Rational Behavior Model*, Technical Report NPSCS-93-003, Naval Postgraduate School, Monterey, California, April 1993.

[141] Clocksin, W. F. and C. S. Mellish, *Programming in Prolog*, 3d ed., Springer-Verlag, 1987.

[142] Culbert, C., G. Riley, and R. T. Savely, "Approaches to the Verification of Rule-Based Expert Systems", *Proceedings of the First Annual Workshop on Space Operations, Automation, and Robotics (SOAR '87)*, pp. 191-196a, August 5-7, 1987, Houston, Texas.

[143] *Autonomous Control Logic Phase II Contract Kickoff Presentation*, July 24, 1991, Lockheed Missiles and Space Company, Palo Alto, California.

[144] *Autonomous Underwater Vehicle Controller Information Brief*, November 1, 1990, Texas A&M University.

[145] Booch, G., *Software Engineering with Ada*, 2d ed, Benjamin/Cummings, 1987.

[146] Giralt, G., R. Chatila, and M. Vaisset, "An Integrated Navigation and Motion Control System for Autonomous Multisensory Mobile Robots", in *The First International Symposium on Robotics Research*, M. Brady and R. Paul, eds., pp. 191-214, MIT Press, 1984.

[147] Kwak S. H., R. B. McGhee, and T. E. Bihari, *Rational Behavioral Model: A Tri-Level Multiple Paradigm Architecture for Robot Vehicle Control Software*, Technical Report NPSCS-92-003, Naval Postgraduate School, Monterey, California, March 1992.

[148] Wang, F., et al., "A Petri-Net Coordination Model of Intelligent Mobile Robots", *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 21, No. 4, pp. 777-789, July/August 1992.

[149] Connell, J., "SSS: A Hybrid Architecture Applied to Robot Navigation", *Proc. of the 1992 IEEE International Conference on Robotics and Automation*, pp. 2719-2724, Nice, France, May 12-14, 1992.

[150] Coste-Manière, E., B. Espiau, and E. Rutten, "Task-Level Robot Programming Combining Object-Oriented Design and Synchronous Approach: A Tentative Study", *Rapports de Recherche No. 1441, Institut National de Recherche en Informatique et en Automatique*, June 1991.

[151] Bellingham, J. G., et al., "Keeping Layered Control Simple", *Proceedings of the 1990 Symposium on Autonomous Underwater Vehicle Technology*, pp. 3-8, Washington, D.C., June 5-6, 1990.

[152] Personal conversation between J. G. Bellingham and R. B. Byrnes, April, 1992.

[153] Albus, J. S., *System Description and Design Architecture for Multiple Autonomous Undersea Vehicles*, NIST Technical Note No. 1251, September, 1988.

[154] Chatila, R., R. Alami, and R. Prajoux, "An Architecture Integrating Task Planning and Reactive Execution Control", *Proceedings of the Workshop on Architectures for Intelligent Control, IEEE International Conference on Robotics and Automation*, pp. 16-22, Nice, France, May 10, 1992.

[155] MacPherson, D. L., *Spatial Learning by an Autonomous Mobile Robot Using Ultrasonic Range-Finders*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, California, September 1993 (draft).

[156] Marques, D., et al., "Easy Programming: Empowering People to Build Their Own Applications", *IEEE Expert*, Vol. 7, No. 3, pp. 16-29, June 1992.

[157] *Staff Organization and Operations*, FM 101-5, Department of the Army, May 1984.

[158] Knaus, R., "Go with the Flow", *AI Expert*, Vol. 7, No. 6, pp. 17-19, June 1992.

[159] Brownston, L., et al., *Programming Expert Systems in OPS5*, Addison-Wesley, 1985.

[160] *CLIPS Reference Manual*, JSC-22948, Software Technology Branch, Johnson Space Center, NASA, January 11, 1991.

[161] Berzins, V. and Luqi, *Software Engineering with Abstractions*, Addison-Wesley, 1991.

[162] Cox, I. J., et al., "Real-Time Software for Robotics", *AT&T Technical Journal*, pp. 61-72, March/April 1988.

[163] Jensen, E. D., "Asynchronous Decentralized Realtime Computer Systems", *NATO Advanced Study Institute on Real-Time Computing*, October 5-18, 1992, Sint Maarten (preprint).

[164] Ogata, K., *Modern Control Engineering*, Prentice Hall, 1970.

[165] Yoerger, D. R., J. G. Cooke, and J. E. Slotine, "The Influence of Thruster Dynamics on Underwater Vehicle Behavior and Their Incorporation into Control System Design", *IEEE Journal of Oceanic Engineering*, Vol. 15, No. 3, pp. 167-178, July 1990.

[166] McGhee, R. B., A. J. Healey, and S. H. Kwak, *An Experimental Study of Software Architectures and Software Reuse for Control of Unmanned Underwater Vehicles*, Research Proposal to the National Science Foundation, Department of Computer Science, Naval Postgraduate School, Monterey, California, December 1992.

[167] Sha, L. and J. B. Goodenough, "Real-Time Scheduling Theory and Ada", *Computer*, Vol. 23, No. 4, pp. 53-63, April 1992.

[168] Papoulias, F. A. and S. R. Chism, "Path Keeping of Autonomous Underwater Vehicles Using Sliding Mode Control", *International Shipbuilding Progress*, Vol. 39, No. 419, pp. 215-246, September 1992.

[169] Wong, H. C. and D. E. Orin, "Reflex Control of the Prototype Leg During Contact and Slippage", *Proceedings of the 1988 International Conference on Robotics and Automation*, pp. 808-813, April 24-29, 1988, Philadelphia, Pennsylvania.

[170] *GESMOS-68 OS-9/68000 Operating System*, Ver. 2.3, GESPAC, Inc., Geneva, SA, 1988.

[171] Healey, A. J., et al., "Mission Planning, Execution, and Data Analysis for the NPS AUV II Autonomous Underwater Vehicle", *Proceedings of the First IARP Workshop on Mobile Robots for Subsea Environments*, pp. 177-186, Monterey, California, October 23-26, 1990.

[172] Bihari, T. E. and P. Gopinath, "Object-Oriented Real-Time Systems: Concepts and Examples", *Computer*, Vol. 25, No. 12, pp. 25-32, December 1992.

[173] Ishikawa, Y., H. Tokuda, and C. W. Mercer, "An Object-Oriented Real-Time Programming Language", *Computer*, Vol. 25, No. 10, pp. 66-73, October 1992.

[174] Waldron, K. J. and R. B. McGhee, "The Adaptive Suspension Vehicle", *IEEE Controls Magazine*, Vol. 6, No. 6, pp. 7-12, December, 1986.

[175] Goheen, K. R., and E. R. Jefferys, "Multivariable Self-Tuning Autopilots for Autonomous and Remotely Operated Vehicles", *IEEE Journal of Oceanic Engineering*, Vol. 15, No. 3, pp. 144-151, July, 1990.

[176] Healey, A. J. and D. Lienard, "Multivariable Sliding Mode Control for Autonomous Diving and Steering of Unmanned Underwater Vehicles", to appear in *IEEE Journal of Oceanic Engineering*, 1993.

[177] Armitage, M., *Unmanned Aircraft*, Brassey's, 1988.

[178] *Quintus Prolog Manual*, Release 3.1, Quintus Corporation, 1991.

[179] Cameron, C. B., *Control of an Experiment to Measure Acoustic Noise in the Space Shuttle*, M.S.E.E. Thesis, Naval Postgraduate School, Monterey, California, June 1989.

[180] *VxWorks Promotional Guide*, Wind River Systems, Inc., 1010 Atlantic Avenue, Alemeda, California.

[181] Metcalfe, R. and D. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM*, Vol. 19, No. 7, July 1976.

[182] Tanenbaum, A. S., *Computer Networks*, 2d ed., Prentice-Hall, 1988.

[183] Steer, B., S. Dunn, and S. Smith, *Advancing and Assessing Autonomy in Underwater Vehicle Technology Through Inter-Institutional Competitions and/ or Cooperative Demonstrations*, Department of Ocean Engineering, Florida Atlantic University, Boca Raton, Florida, May 1992.

[184] Nordman, D. B., *A Computer Simulation Study of Mission Planning and Control for the NPS Autonomous Underwater Vehicle*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1989.

[185] Wilkinson, W. P., *Mission Executor for an Autonomous Underwater Vehicle*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1991.

[186] Stavridis, J., *Watch Officer's Guide*, 13th ed., Naval Institute Press, 1992.

[187] Kwak, S. H., et al., "Incorporation of Global Positioning System into Autonomous Underwater Vehicle Navigation", *Proceedings of the 1992 Symposium on Autonomous Underwater Vehicle Technology (AUV '92)*, pp. 291-297, Washington, D. C., June 2-3, 1992.

[188] Compton, M. A., *Minefield Search and Object Recognition for Autonomous Underwater Vehicles*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1992.

[189] Ong, S. M., *A Mission Planning Expert System with Three-Dimensional Path Optimization for the NPS Model 2 Autonomous Underwater Vehicle*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1990.

[190] Brutzman, D. P., "From Virtual World to Reality: Designing an Autonomous Underwater Vehicle", paper presented at the *AAAI Fall Symposium on Applications of Artificial Intelligence to Real World Autonomous Mobile Robots*, Cambridge, Massachusetts, October 23-25, 1992.

[191] Zyda, M. J., et al. "Three-Dimensional Visualization of Mission Planning and Control for the NPS Autonomous Underwater Vehicle", *IEEE Journal of Oceanic Engineering*, Vol. 15, No.3, pp. 217-221, July 1990.

[192] Firby, R. J., *Adaptive Execution in Complex Dynamic Worlds*, Technical Report YALEU/CSD/RR#672, Yale University, 1989.

[193] Bonasso, R. P., D. R. Yoerger, and W. K. Stewart, "Semi-Autonomous Vehicles for Shallow Water Mine-Clearing", *Proceedings of the 1992 Symposium on Autonomous Underwater Vehicle Technology (AUV '92)*, pp. 22-28, Washington, D. C., June 2-3, 1992.

[194] Hultman, J., A. Nyberg, and M. Svensson, "A Software Architecture for Autonomous Systems", *Sixth International Symposium on Unmanned Untethered Submersible Technology*, Durham, New Hampshire, 1989.

[195] Hayes-Roth, B., "Architectural Foundations for Real-Time Performance in Intelligent Systems", *Journal of Real-Time Systems*, Vol. 2, No. 1/2, 1990.

[196] Kwak, S. H. and R. B. McGhee, *Rule-Based Motion Coordination for the Adaptive Suspension Vehicle on Ternary-Type Terrain*, Technical Report NPSCS-91-006, Naval Postgraduate School, Monterey, California, December 1990.

[197] Schultz, A. C., J. J. Grefenstette, and K. A. De Jong, "Adaptive Testing of Controllers for Autonomous Vehicles", *Proceedings of the 1992 Symposium on Autonomous Underwater Vehicle Technology (AUV '92)*, pp. 158-164, Washington, D. C., June 2-3, 1992.

[198] Safford, D. R., *FTMP: A Protocol for Operating System Fault Tolerance in a Fully Distributed, Loosely Coupled Environment*, Ph. D. Dissertation, Texas A&M University, College Station, Texas, December 1990.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Dudley Knox Library Code 52 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Chairman, Department of Computer Science Code CS Naval Postgraduate School Monterey, California 93943	1
4.	Computer Technology Programs Code 37 Naval Postgraduate School Monterey, California 93943	1
5.	Professor R. B. McGhee, Code CS/Mz Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
6.	Professor Luqi, Code CS/Lq Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
7.	Professor S. Shukla, Code EC/Sh Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943	1
8.	Professor A. J. Healey, Code ME/Hy Department of Mechanical Engineering Naval Postgraduate School Monterey, California 93943	1

9. MAJ M. L. Nelson — 1
USCINCPAC/J6612
Box 32A, Bldg. 35
Camp H. M. Smith, HI 96861

10. Professor S. H. Kwak, Code CS/Kw — 1
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943

11. Professor S. R. Garrett, Code PH/Gx — 1
Department of Physics
Naval Postgraduate School
Monterey, CA 93943

12. Mr. Norman Caplan — 1
Biological and Critical Systems Division
Engineering Directorate
National Science Foundation
1800 G Street, NW
Washington, D. C. 20550

13. MAJ R. B. Byrnes, Jr. — 1
Software Technology Branch, ARL
115 O'Keefe Building
Georgia Institute of Technology
Atlanta, GA 30332-0800